NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SOFTWARE ARCHITECTURE FOR DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS
by
Jose Carlos Alves de Almeida
September 1998

Thesis Advisor:	Man-Tak Shing
Co-Advisor:	Michael Holden

Approved for public release; distribution is unlimited.

19981110 139

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
SOFTWARE ARCHITECTURE FOR DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

5. FUNDING NUMBERS

6. AUTHOR(S)
Almeida, Jose Carlos Alves de

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Real-time embedded systems have particularly strict requirements on accuracy, safety and reliability. A central question in the design of such systems is how to support concurrent processing without adversely affecting the timing requirements of the system. Concurrent processing is essential because the only way to successfully meet some tight real-time constraints is to use multiple processors.

This thesis focuses on the distributed scheduling problem. It proposes a distributed scheduling algorithm to allocate and schedule a set of tasks onto a collection of processors linked by a network. It further proposes a distributed software architecture for CAPS (Computer Aided Prototyping System) generated prototypes based on GLADE (GNAT Library for Ada Distributed Execution).

The new distributed CAPS architecture is applied to several prototype examples. The results show that it is possible to build distributed real-time embedded systems under the distributed scheduling model, where sets of tasks run independently on each processor, using GLADE.

14. SUBJECT TERMS
Real-Time Embedded Systems, Distributed Systems, Real-Time Scheduling, Software Architecture, Computer Aided Prototyping.

15. NUMBER OF PAGES
171

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-9)
Prescribed by ANSI Std. 239-18

SOFTWARE ARCHITECTURE FOR DISTRIBUTED REAL-TIME EMBEDDED SYSTEMS

Jose Carlos Alves de Almeida
Lieutenant, Brazilian Navy
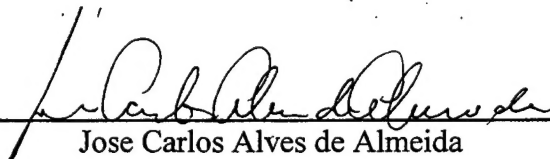B.S., Brazilian Naval Academy, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

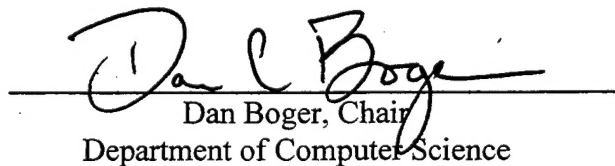## NAVAL POSTGRADUATE SCHOOL
September 1998

Author: _____
Jose Carlos Alves de Almeida

Approved by: _____
Man-Tak Shing, Thesis Advisor

_____
Michael Holden, Co-Advisor

_____
Dan Boger, Chair
Department of Computer Science

ABSTRACT

Real-time embedded systems have particularly strict requirements on accuracy, safety and reliability. A central question in the design of such systems is how to support concurrent processing without adversely affecting the timing requirements of the system. Concurrent processing is essential because the only way to successfully meet some tight real-time constraints is to use multiple processors.

This thesis focuses on the distributed scheduling problem. It proposes a distributed scheduling algorithm to allocate and schedule a set of tasks onto a collection of processors linked by a network. It further proposes a distributed software architecture for CAPS (Computer Aided Prototyping System) generated prototypes based on GLADE (GNAT Library for Ada Distributed Execution).

The new CAPS architecture is applied to several prototype examples. The results show that it is possible to build distributed real-time embedded systems under the distributed scheduling model, where sets of tasks run independently on each processor, using GLADE.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENT

I would like to express my gratitude to my beloved wife, Danielle, who firmly and patiently stimulated me, as well as, inspired me with courage and hope throughout the whole period I was preparing and, later, writing this thesis. Next, I would like to thank my mother Wilma for her devotion, love and support throughout my life.

To my thesis advisor, Professor Man-Tak Shing, I would like to thank him for all of his assistance, guidance and support throughout the thesis process. I would also like to thank my thesis co-advisor, Commander Michael Holden, USN, for his support.

In addition, I would like to thank the staff of the Computer Science Department, especially Valerie Brooks and Jean Brennan for their support.

Finally, I would like to thank God for helping me to advance one more step in my life and my career.

I. INTRODUCTION

This thesis deals with the area of computer-aided real-time systems development. Here we investigate the capabilities of distributed real-time systems support in Ada 95 and the issues involved in automating software development for such systems. We divided this work into seven chapters. This chapter presents the basic concepts involving real-time embedded systems, introduces the reader to the real-time scheduling problem and explains the importance of computer-aided prototyping systems. In Chapter II, we present a distributed system overview. Chapter II is considered very important because it explains the basic concepts behind the implementation of the Ada 95 Distributed Systems Annex and will help the reader to get a better understanding of the following chapters.

In Chapter III, we explore the characteristics of the Prototyping System Description Language (PSDL) real-time model and discuss a number of issues related to the distributed system model. Chapter IV is an extension of the previous chapter. There we discuss the distributed scheduling problem and propose a technique to allocate and schedule tasks running on a network of processors in a distributed system.

In Chapter V, we discuss the Ada 95 Distributed Systems Annex and present the Gnat Library for Ada Distributed Execution (GLADE), the GNAT implementation of Annex E (Ada, 1995).

Chapter VI presents the current CAPS (Computer Aided Prototyping Systems) uniprocessor architecture and the proposed distributed implementation. In Chapter VII, we conclude this work.

In this thesis, our concern is to identify the requirements for distributed real-time embedded systems and discuss the merits of a number of approaches. Of these, the design and development of an Ada 95 software architecture for distributed real-time embedded systems and automatic generation tools for such architecture is the most significant.

A. REAL-TIME EMBEDDED SYSTEMS

For most programs, correctness depends only on the logical sequence in which instructions are executed, not when they are executed. In contrast, real-time systems interact with the external world in a way that involves time. When a stimulus appears,

1

the system must respond to it in a certain way and before a certain deadline. If it delivers the correct answer but after the deadline, the system is regarded as having failed (Tanenbaum, 1995).

A real-time system is a system that must satisfy explicit (bounded) response time constraints or risk severe consequences, including failure (Laplante, 1993). Failure means that the system can not satisfy the requirements specified in the formal system specification. Real-time embedded systems are those used to control specialized hardware in which the computer system is installed. For example, the microprocessor system used in the cruise control system of many automobiles is an embedded system. Similarly, the software used to control the inertial guidance system of a space shuttle or the operation of an assembly line in an industrial plant is embedded because it operates in a highly specialized hardware environment.

A space shuttle must process accelerometer data within a certain period of time, which depends on the specification of the space shuttle. Failure to do so could result in a false position or velocity indication and cause the space shuttle, at best, to go off-course; or at the worst to crash. For a steam generator low water problem, failure to respond swiftly could result in severe damage to the equipment and loss of life. The examples mentioned earlier satisfy the criteria for a real-time system. In short, a system does not have to process data in microseconds to be considered real-time, it must simply have response times that are constrained and thus predictable (Laplante, 1993). One of the most important properties of any real-time system is that its behavior be predictable. It should be clear at design time that the system can meet all of its deadlines, even in the worst case condition.

Real-time embedded systems support various aspects of modern life. The increased power of microprocessors and steadily falling prices have made digital control systems technically attractive and highly cost-effective. Television sets, cars, instruments and telecommunication equipment are controlled by microprocessors and the related software, which is embedded into the product itself. The user does not see or feel this software. The only indication of its existence is the large set of operations provided by the product.

Real-time systems are classified as either hard real-time systems or soft real-time systems. In hard real-time systems an early and late response are both treated as an error and may cause damage or loss of life or property. Hard time constraints appear often in control applications, such as sophisticated fly-by-wire systems in aircraft, arm controllers in industrial robots, or anti-lock braking systems in cars.

In soft real-time systems, a late response is normally acceptable. Examples of soft real-time systems are communications equipment, such as digital telephone exchanges. Performance issues are critical in real-time systems. In the case of soft real-time systems, performance considerations are related to the capability to adequately handle the external load on the system. In hard real-time systems, performance requirements mean the ability to meet all specified deadlines (Awad, Kuusela and Ziegler, 1996).

As noted, all practical systems represent at least soft real-time systems. Since we are most interested in hard real-time systems, we will use the term "real-time system" to mean hard real-time system without loss of generality.

Typically, a real-time system has to perform several different tasks. Some of these tasks are periodic, that is, they are executed at regular time intervals. Other tasks are aperiodic, that is, the need to execute a task may occur at any arbitrary point in time. A real-time system that is able to react to aperiodic request within time limits is called reactive. Most embedded real-time systems are reactive, that is, they have to be able to react to new events, even if the system is still processing earlier tasks. Thus, competing requests are processed concurrently.

In addition to time constraints, a task can have other constraints such as (Stankovic and Ramamritham, 1988):

1. Resource constraints - the resources required during the execution of the task
2. Precedence constraints - that specify a partial (perhaps total) ordering on the execution of tasks
3. Concurrency constraints - that describe which tasks can run concurrently, to share a resource, for example
4. Placement constraints - whether a given task is to run in a specific processor
5. Criticalness - the relative value to the system that is associated with some specific task when it meets its deadline
6. Preemptiveness - determining whether a task can be interrupted by other tasks and resume execution afterwards

3

7. Communication requirements - issues of inter-task communications and synchronization protocols such as acceptable delays.

Since embedded systems are connected to the surrounding real world, processor overload may occur. In an overload situation, the performance degradation of the system should take place gracefully. During the shortage of resources caused by an overload situation, some tasks will have to wait for processing.

Tasks are classified as critical, essential, and nonessential. Critical tasks have deadlines that must be met. Essential tasks also have deadlines, but failure to meet them will not cause severe problems. Nonessential tasks are allowed to wait without any specified time limit. Task scheduling in hard real-time systems can be either static or dynamic. In static scheduling, it is assumed that all information about the task is known *a priori,* and the schedule is usually generated off-line. In dynamic scheduling, although all information about the tasks may be known *a priori,* they are allowed to be dynamically invoked, and the schedule is calculated "on the fly." There has been a great deal of debate about the appropriateness of dynamic scheduled algorithms for hard real-time systems. Many people are in favor of static scheduling because it seems reasonable to assume that for safety-critical applications all the schedulability should be guaranteed before execution (Audsley and Burns, 1993).

B.   REAL-TIME SCHEDULING

Real-time systems are frequently programmed as a collection of short tasks, each with a well-defined function and a well-bounded execution time. The response to a given stimulus may require multiple tasks to be run, generally with constraints on their execution order. In addition, a decision has to be made about which tasks to run on which processors (Tanenbaum, 1995).

Scheduling can be centralized, with one machine collecting all the information and making all the decisions, or it can be decentralized, with each processor making its own decisions. In the centralized case, the assignment of tasks to processors can be made at the same time, whereas in the decentralized case assigning tasks to processors is distinct from deciding which of the tasks assigned to a given processor to run first. A key question that all real-time system designers face is whether or not it is even possible to meet all the constraints.

4

Real-time scheduling algorithms can be characterized by the following parameters:

1. Preemptive versus non-preemptive scheduling.
2. Dynamic versus static.
3. Centralized versus decentralized.

1. Preemptive Scheduling

Preemptive scheduling allows a task to be suspended temporarily when a higher priority task arrives, resuming it later when no higher priority task is available to run. Non-preemptive scheduling runs each task to completion. Once a task is started, it continues to hold the processor until it is done.

2. Dynamic Scheduling

Dynamic algorithms make their scheduling decisions during execution. When an event is detected, a dynamic preemptive algorithm decides on the fly whether to run the first task associated with the event or to continue running the current task. When the current task finishes, a choice is made among the ready tasks.

The classic scheduling algorithm is the Rate Monotonic Algorithm (Liu and Layland, 1973). It was designed for preemptively scheduling periodic tasks on a single processor. Each task is assigned a priority equal to its execution frequency. The higher the execution frequency, the higher the priority. At run time, the scheduler always selects the highest priority task to run, preempting the current task if it is needed.

A second preemptive dynamic algorithm is Earliest Deadline First. Whenever an event is detected, the scheduler adds it to the list of waiting tasks, which is kept sorted by the deadline, with the task with the closest deadline first. Then the scheduler chooses the first task in the list, the one closest to its deadline. Liu and Leyland (1973) proved that the Earliest Deadline First algorithm is optimal for any set of independent periodic tasks.

3. Static Scheduling

Static scheduling is done before the system starts operating. The input consists of a list of all tasks and the times that each must run. The goal is to find an assignment of tasks to processors and for each processor, a static schedule giving the order in which the tasks are to be run. In theory, the scheduling algorithm can run an exhaustive search to find the optimal solution, but the search time is exponential to the number of tasks (Ullman, 1976).

5

Let us assume that every time an event is detected, a task is started on processor A. This task, in turn, can start up additional tasks on both processor A or processor B, and so on until the last task is executed, as illustrated in Figure 1.1. We consider that a task can not start until a message from another task has arrived. The scheduler analyzes the graph of Figure 1.1, using as input the information about the running times of all the tasks, and then applies some heuristics to find a good schedule. Two potential schedules are given in Figure 1.2 a) and b). Messages between tasks on different processors are depicted as arrows; messages between tasks on the same machine are handled internally and are not shown.
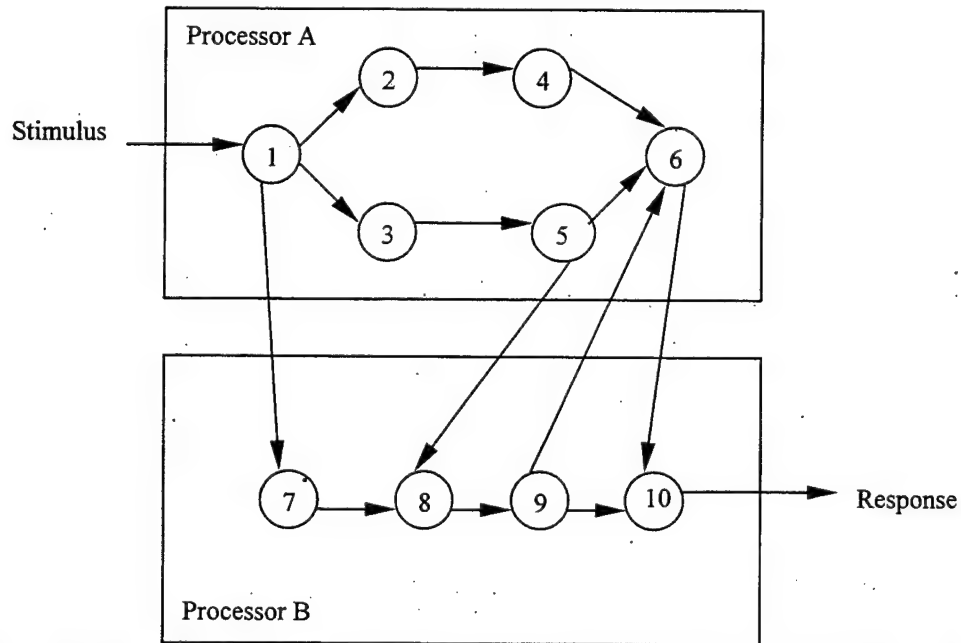
Figure 1.1. Ten Real-Time Tasks to be Executed on Two Processors (Tanenbaum, 1995)

Of the schedules illustrated, the one in Figure 1.2 (b) is a better choice because it allows task 5 to run early, thus making it possible for task 8 to start earlier. If task 5 is delayed significantly, as in Figure 1.2 (a), then tasks 8 and 9 are delayed, which also means that 6 and eventually 10 are delayed, too.
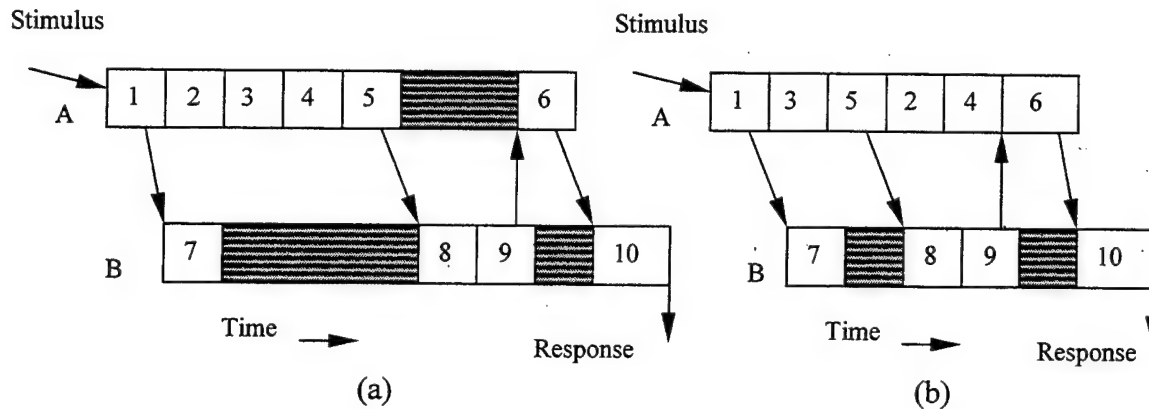
Figure 1.2. Two Possible Schedules for the Tasks of Figure 1.1

When we take communication into account, the problem of scheduling the same tasks would change due to the delay introduced by the communication subsystem. Starting a task would require a delay that was not present before. The important thing to notice about this example is that the run time behavior is completely deterministic and known even before the program starts executing. As long as communication and processor errors do not occur, the system will always meet its real-time deadlines.

The choice of dynamic or static scheduling is an important one and has far-reaching consequences for the system. Static scheduling must be carefully planned in advance, with considerable effort going into choose the various parameters. Dynamic scheduling does not require as much advance work, since scheduling decisions are made on the fly, during execution (Tanenbaum, 1995).

C.   CONCURRENCY ASPECTS IN REAL-TIME EMBEDDED SYSTEMS

Concurrency aspects always occur in real-time embedded systems. The reason is that concurrency is an inherent feature of real-time applications, and must be included in every modeling effort. External events may occur at any point in time, even simultaneously, and they must be queued and handled within preset time limitations (Awad, Kuusela and Ziegler, 1996).

A central question in the design of real-time embedded systems is how to support concurrent processing without affecting the timing requirements. Concurrent processing is essential because the only way to make some tight real-time constraints feasible is to use multiple processors.

Concurrency is a powerful concept that solves some problems of real-time systems. On the other hand, it creates new problems regarding the consistency of data, because the same data can be simultaneously accessed by two concurrent requests on the same object. Also, we have to deal with other problems inherent to distributed systems: synchronization, resource management, and communication with external systems and processes. In the following chapters, we will see how to handle concurrency in distributed real-time embedded systems.

D. COMPUTER AIDED PROTOTYPING SYSTEMS

Rapid prototyping can be used to reduce the risks of producing systems that do not meet the customers needs (Luqi, 1993). A prototype is an executable pilot version of a proposed software system. Prototypes are used to gain information that can guide analysis and design, and can support automatic generation of production code.

Real-time and embedded systems have particularly strict requirements on accuracy, safety and reliability, and are usually subject to timing constraints that must be met even under the worst possible operating conditions. Since feasible requirements for large embedded systems are difficult to formulate, understand and meet without extensive prototyping, computer aid is the key to rapid construction, evaluation and evolution of such prototypes.

The Computer Aided Prototyping Systems (CAPS) can be used to prototype large, parallel, real-time and distributed systems because the requirements for such software systems are difficult to assess. CAPS is a set of software tools, which provide a means to validate functional requirements and verify design specifications early in the development of the software system. It implements the rapid prototyping concept via a high-level prototyping language called PSDL. This language is designed for prototyping real-time and large software systems, and supports conceptual modeling of such systems (Luqi, 1993).

The prototyping language PSDL supports a modeling strategy based on data flow graphs augmented with non-procedural timing and control constraints. The real-time aspects of PSDL are described in Chapter III.

II.   DISTRIBUTED SYSTEMS OVERVIEW

A.   DISTRIBUTED SYSTEMS CHARACTERISTICS

A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed systems software. Distributed systems software enables computers to coordinate their activities and to share the resources of the system: hardware, software and data. From the point of view of users, the distributed system should act as a single, integrated computing facility even though they are aware that distinct machines connected by a communication subsystem are being used.

The definition above corresponds to a type of distributed system known as loosely-coupled systems, where the shared resources needed to provide an integrated computing service are provided by some of the computers in the network and are accessed by system software that runs on all of the computers, using the network to coordinate their work and to transfer data between them.

The other type, known as tightly-coupled systems, exploit multiple processing units, often sharing a single memory or address space, to achieve high performance in a computer system that is otherwise centralized.

In this thesis, we are concerned with the architecture and design of general purpose distributed systems, and we will use the term "distributed systems" to mean loosely-coupled systems.

Six key characteristics are primarily responsible for the usefulness of distributed systems. They are not automatic consequences of distribution. System and application software must be carefully designed in order to ensure that their desired usefulness is attained.

1.   Resource Sharing

The term resource may be defined as the range of things that can be shared usefully in a distributed system. This includes hardware components such as storage devices, printers, and software-defined entities such as files, databases and other data objects. Resources in a distributed system are physically encapsulated within one of the computers and can only be accessed from other computers by communication. For effective sharing each resource must be managed by a program that offers a

11

communication interface enabling the resource to be accessed, manipulated, and updated reliably and consistently (Coulouris, Dollimore and Kindberg, 1996).

Each type of resource requires specific management policies, but there are also common requirements. These include the provision of a naming scheme for each class of resource that enables individual resources to be accessed from any location, the mapping of resource names to communication addresses, and the coordination of concurrent accesses that change the state of the shared resources in order to ensure their consistency.

We can imagine a distributed system as a system composed of a set of resource managers and a set of resource using programs. The resource users communicate with the resource managers to access the shared resources of the system. This perspective leads to the development of an interesting model for distributed systems, the client-server model. Figure 2.1 illustrates the client-server model.
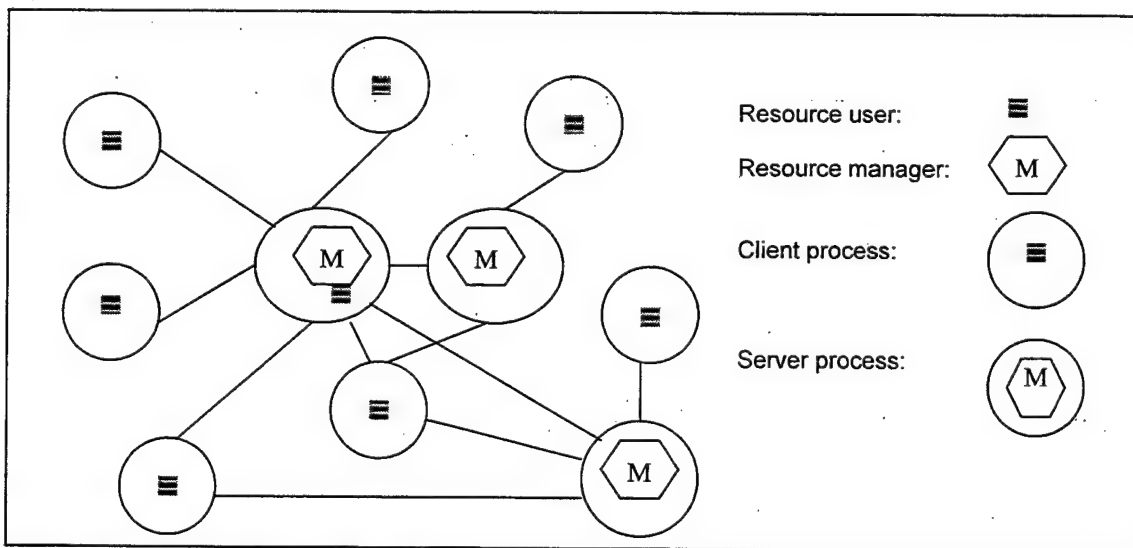
Figure 2.1. Client-Server Model

There is a set of server processes, each acting as a resource manager for a collection of resources of a given type, and a set of client processes, each executing a task that requires access to some shared resources. Resource managers may themselves need to access shared resources managed by other processes, so some processes are both client and server processes. In the client-sever model, all shared resources are held and managed by server processes. Client processes send a message to the server processes requesting access to one of their resources when they are needed. If the request is a valid

12

one, then the server executes the request and sends back a reply to the client that contains the result of the requested processing.

The client-server model provides an effective general-purpose approach to sharing of information and resources in distributed systems. The model can be implemented in a variety of different hardware and software environments. The computers used to run the client and server processes can be of many types and there is no need to distinguish between them. It is even possible for both the client and server processes to be run on the same computer. Moreover, some processes are both client and server processes. That is, a server process may use the services of another server, appearing as a client to the latter (Sinha, 1997). It should be clear that server processes are not centralized providers of the resources they manage. Also, a distinction is made between the services that are provided to clients and the servers that provide them. A service is considered to be an abstract entity that may be provided by several server processes running on separate computers and cooperating via the network.

2. Openness

The openness of a computer system is the characteristic that determines whether the system can be extended in various ways. For example, adding additional resources like processors, communication interfaces, or the addition of software extensions such as operating system features, communications protocols, and resource-sharing services. The openness of a distributed system is determined primarily by the degree to which new resource sharing services can be added without disruption to or duplication of existing services.

3. Concurrency

When several processes exist in a single computer, we say that they are executed concurrently. This is achieved by interleaving the execution of portions of each process. In distributed systems there are many computers, each with one or more central processors. If there are N computers in a distributed system with one processor each, then up to N processes can run in parallel, assuming that each process is located in a different computer. Concurrency and parallel execution arise naturally in distributed systems from the separate activities of users, the independence of resources, and the location of server processes in separate computers. The separation of these activities

enables processing to proceed in parallel in separate computers. Concurrent accesses and updates to shared resources must be synchronized (Coulouris, Dollimore and Kindberg, 1996).

4. Scalability

Distributed systems operate effectively and efficiently at many different scales. The range of computers in a distributed system extends from the smallest practicable distributed system consisting of two workstations and a file server to hundreds of workstations and many file servers connected via a local-area network (LAN). The system and application software should not need to change when the scale of the system increases. However, further research is required in this area to accommodate the very large-scale systems and applications that will probably emerge as inter-networking increases and high performance networks appear.

5. Fault Tolerance

When faults occur in hardware or software, programs may produce incorrect results or may stop before completing the intended computation. The design of fault-tolerant computer systems is based on two approaches, both of which must be deployed to handle each fault:

1. Hardware redundancy;
2. Software recovery.

To produce systems that are tolerant to hardware failures, we should implement them using redundant components for the critical services. For example, we could implement the file service of a distributed system as a group of file servers that closely cooperate with each other to manage the files of the system and work in such a manner that the system will continue to operate even if only one file server is up and working.

Software recovery involves the design of software so that the state of permanent data (files and other material stored in permanent storage) can be recovered when a fault is detected. Some of the commonly used techniques for implementing this method in a distributed system are as follows:

1. Atomic transactions: an atomic transaction is a computation consisting of a collection of operations that take place indivisibly in the presence of failures and concurrent computations. That is, either all of the operations are performed successfully or none of their effects prevails, and other processes

14

executing concurrently can not modify or observe intermediate states of the computation.

2. Stateless servers: the client-server model is frequently used in distributed systems to service user requests. In this model, a server may be implemented by using either the "stateful" or "stateless" service paradigm. The distinction between them is whether or not the history of the serviced requests affects the execution of the next service request. The stateful approach depends on the history of the serviced requests, but the stateless does not depend on it. In the event of a failure, the stateless service paradigm makes crash recovery easier because no client information is required to be maintained by the server.

3. Acknowledgments and time-out based retransmissions of messages: in a distributed system, events such as a node crash or a communication link failure may interrupt a communication that was in progress between two processes, resulting in the loss of a message. Therefore, a reliable interprocess communication mechanism must have ways to detect lost messages so that they can be retransmitted.

As expected, the implementation of these techniques should be carefully examined. According to Sinha (1997), the mechanisms described above may be employed to create a very reliable distributed system. However, the main drawback of increased system reliability is the potential reduction of execution time efficiency from the extra overhead involved in these techniques.

6. Transparency

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than a collection of independent components. The implications of transparency are a major influence on the design of the system software (Coulouris, Dollimore and Kindberg, 1996).

The separation of components is an inherent property of distributed systems. Therefore, a communication subsystem and explicit system management and integration techniques is needed. Nevertheless, the distributed system should allow users to access remote resources in the same way as local resources. That is, the user interface, which takes the form of a set of system calls, should not distinguish between local and remote resources. It should also be the responsibility of the distributed system to locate the resources and to arrange for servicing user requests in a user-transparent form.

The International Standards Organization's Reference Model for Open Distributed Processing (International Standards Organization, 1992) identifies eight forms of

transparency. We use the term "object" to denote the entities to which distribution transparency is applied:

1. Access transparency enables local and remote objects to be accessed using identical operations.
2. Location transparency enables objects to be accessed without knowledge of their location.
3. Concurrency transparency enables several processes to operate concurrently using shared information objects without interference between them.
4. Replication transparency enables multiple instances of objects to be used to increase reliability and performance without knowledge of the replicas by users or application programs.
5. Failure transparency enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.
6. Migration transparency allows the movement of objects within a system without affecting the operation of users or application programs.
7. Performance transparency allows the system to be configured to improve performance as loads vary.
8. Scaling transparency allows the system and applications to expand in scale without change to the system structure or the application algorithms.

B. ARCHITECTURAL ISSUES

In a distributed environment, it is likely to encounter a heterogeneous network of computers with different clock speeds, CPUs, memory systems, etc. It is therefore important to realize how these attributes can affect the overall performance and functionality of the system. The following section will introduce some of the problems that can affect the design of distributed systems, especially distributed real-time systems, so that the reader may be aware of their existence and importance.

1. Different Clocks

A distributed system consists of several nodes, each with its own clock, running at its own speed. The precision of a clock is directly related to its granularity, the minimum number of ticks it can handle, and the quality of its time reference, which is usually based on a crystal that oscillates at well-defined frequencies. Therefore, the first limit imposed by the clock is the minimum acceptable oscillation period. This is not, in practice, an actual limitation, since typical clocks range from tens to hundreds of megahertz, providing nanosecond minimum allowable periods.

The real problem is that clocks can drift, causing a variety of synchronization problems. Maintaining an accurate global clock is one of the most challenging tasks in

the distributed system arena. Usually this is achieved at the cost of substantial overhead in communications.

2. CPU Speed

A distributed system consists of a collection of distinct processes that are spatially separated and run concurrently in multiple processors. Due to the multiplicity of processors, distributed systems are expected to have better performance than single-processor centralized systems. However, when different processors are present, the net result is a different execution time for the same piece of code on each processor. Since the maximum execution time is the most basic timing property for real-time systems and is also taken into account when scheduling the processes, this factor can increase the complexity of the scheduling problem when designing distributed real-time systems.

3. Memory

Again, due to the heterogeneity of distributed systems, the designer of a distributed real-time system should consider issues like cache size, paging, number of pipelining stages, etc., which can affect the overall throughput of the system, and consequently the timing requirements of the application.

4. The Communication Media

This is one of the most important factors when dealing with distributed systems, and can greatly affect final timing requirements for the application. Note also that the timing requirements are affected not only by the actual transmission delay, but also by the operating systems invoked on behalf of the application.

C. DESIGN ISSUES

In Section A, we discussed the key characteristics of distributed systems. In this section, we focus on the system architectures that are used to meet those requirements and the technical issues that must be addressed in their design. The designer of distributed systems or applications must consider a number of issues, such as software structure and software engineering techniques. However, we shall restrict our discussion to design issues that specifically affect the implementation of real-time embedded systems in a distributed system, namely interprocess communication.

When we say that two computers of a distributed system are communicating with each other, we mean that two processes, one running on each computer, are in

17

communication with each other. In a distributed system, processes executing in different computers often need to communicate with each other to achieve some common goal. Therefore, the system should provide interprocess communication (IPC) mechanisms to facilitate such communication activities.

We can enumerate some of the desirable features of a good message-passing system:

1. Simplicity: a message passing system should be simple and easy to use. It should be possible for a programmer to designate the different modules of a distributed application to send and receive messages between them in the simplest as possible without the need to worry about system or network aspects not relevant at the application level.
2. Uniform semantics: in a distributed system the semantics of remote communications should be as close as possible to those of local communications.
3. Efficiency: efficiency is usually a critical issue for a message-passing system. If the message-passing system is not efficient, interprocess communication may become so expensive that application designers will try to avoid them.
4. Reliability: a reliable IPC protocol can cope with failure problems and guarantees the delivery of a message.

Sinha identifies the following important issues to be considered in the design of an IPC protocol for a message-passing system (Sinha, 1997):

1. Who is the sender?
2. Who is the receiver?
3. Is there one receiver or many receivers?
4. Does the sender needs to wait for a reply?
5. What should be done if a catastrophic event such as a node crash or a communication link failure occurs during the course of communication?
6. What should be done if the receiver is not ready to accept the message?
7. If there are several outstanding messages for a receiver, can it choose the order in which to service the outstanding messages?

These issues are addressed by the semantics of the set of communication primitives provided by the IPC protocol, as we shall see below.

1.   Synchronization

A central issue in the communication structure is the synchronization imposed on the communicating processes by the communication primitives. The semantics used for synchronization may be broadly classified as blocking and nonblocking types. A primitive is said to have nonblocking semantics if its invocation does not block the execution of its invoker; otherwise, a primitive is said to be of the blocking type. The

synchronization imposed on the communicating process depends on the type of semantics used for the send and receive primitives.

In the case of a blocking send primitive, after execution of the send statement the sending process is blocked until it receives a reply from the receiver that the message has been received. For a nonblocking send primitive, after execution of the send statement, the sending process is allowed to proceed with its execution as soon as the message has been copied to a buffer.

In the case of a blocking receive primitive, after execution of the receive statement, the receiving process is blocked until it receives a message. For a nonblocking receive primitive, the receiving process proceeds with its execution after execution of the receive statement, which returns control almost immediately just after telling the kernel where the message buffer is.

An important issue in a nonblocking receive primitive is how the receiving process know that the message has arrived in the message buffer. One of the following two methods is commonly used for this purpose:

1. Polling: in this method, a test primitive is provided to allow the receiver to check the buffer status. The receiver uses this primitive to periodically poll the kernel to check if the message is already available in the buffer.
2. Interrupt: in this method, when the message has been placed in the buffer and is ready for use by the receiver, a software interrupt is used to notify the receiving process. This method permits the receiving process to continue with its execution without having to issue unsuccessful test requests. Although this method is highly efficient and allows maximum parallelism, its main drawback is that user-level interrupts make programming difficult (Tanenbaum, 1995).

In a blocking send primitive, the sending process could be blocked forever in a situation where the receiving process has crashed or the sent message has been lost on the network due to a communication failure. To prevent this situation, blocking send primitives often use a timeout value that specifies an interval of time after which the send operation is terminated with an error status. A timeout value may also be associated with a blocking receive primitive to prevent the receiving process from getting blocked indefinitely when the send process has crashed or the expected message has been lost on the network due to a communication failure. When both the send and receive primitives of a communication between two processes use blocking semantics, the communication

is said to be synchronous, otherwise it is asynchronous. That is, for synchronous communication, the sender and the receiver must be synchronized to exchange a message. Figure 2.2 illustrates this mode of communication.

In synchronous communications, the sending process sends a message to the receiving process, then waits for a reply. After executing the receive statement, the receiver remains blocked until it receives the message. On receiving the message, the receiver sends a reply to the sender. The sender resumes execution only after receiving this reply.

Compared to asynchronous communication, synchronous communication is simple and easy to implement. It also contributes to reliability because it assures the sending process that its message has been accepted before the sending process resumes execution. As a result, if the message gets lost or is undelivered, no backward error recovery is necessary for the sending process to establish a consistent state and resume execution (Shatz, 1984). However, the main disadvantage of synchronous communication is that it limits concurrency and is subject to communication deadlocks.
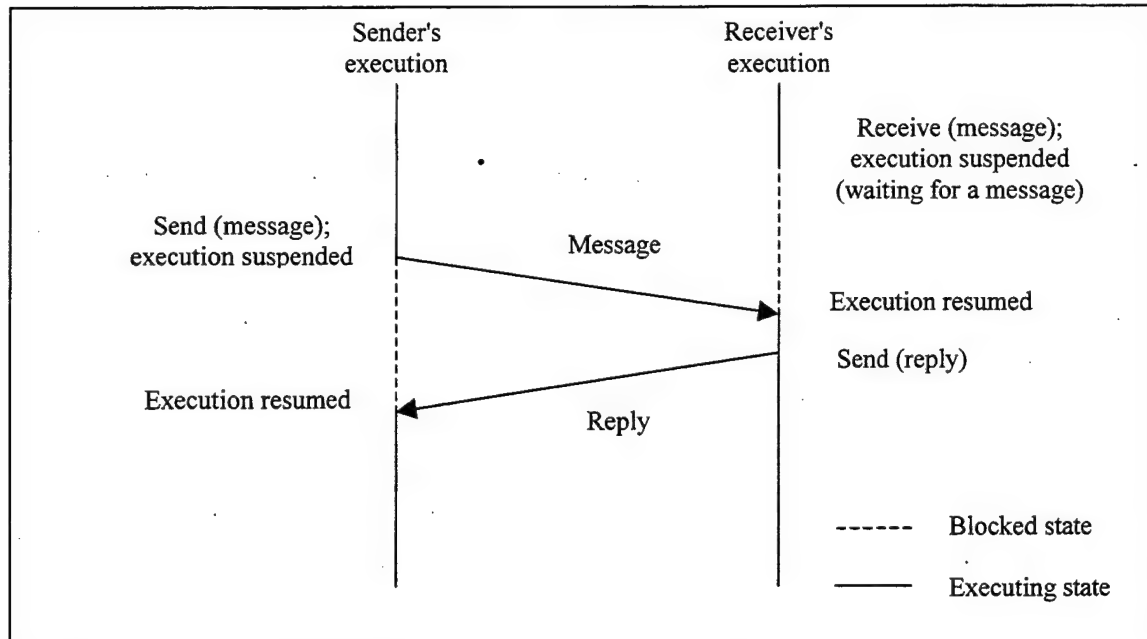
Figure 2.2. Synchronous Mode of Communication

A flexible message-passing system usually provides both blocking and nonblocking primitives for send and receive so that users can choose the most suitable one to match the specific needs of their application.

2.   Naming

Distributed systems are based on the sharing of resources and on the transparency of their distribution. The names assigned to resources or objects must have global meanings that are independent of the location of the object, and they must be supported by a name interpretation system that can translate names in order to enable programs to access named resources. A design issue is to design naming schemes that will scale to an appropriate degree and in which names are translated efficiently to meet appropriate goals for performance (Coulouris, Dollimore and Kindberg, 1996).

A process that requires access to a resource, which it does not manage, must possess a name or a identifier for it. The term "name" refers to names that are interpreted by users or by programs and the term "identifier" refers to names that are interpreted or used only by programs. We say that a name is resolved when it is translated into a form in which it can be used to invoke an action on the resource or object to which it refers. In distributed systems, a resolved name is generally a communication identifier together with other attributes that may be useful for communication. The form of a

21

communication identifier depends on the kinds of identifier interpretation provided in the communication system that is in use. For example, in Internet communication a communication identifier must contain two parts: a host identifier (also called an IP address, the numeric address of a computer, for example 131.120.001.013) and a port number identifying a particular communication port among those located at the host.

The resolution of names may involve several translation steps. At each step, a name or identifier is mapped to a lower-level identifier that can be used to specify a resource when communicating with some software component. At some stage in this sequence of translations a communication identifier is produced that is acceptable to the communication subsystem that is in use, and this can be used to transmit a request to a resource manager. The communication subsystem may have to perform further translations to produce network addresses and routing information that are acceptable to lower-level network software layers (Coulouris, Dollimore and Kindberg, 1996).

In distributed systems a communication subsystem usually supports two types of process addressing:

1. Explicit addressing: the communication primitive explicitly names the process with which communication is desired as a parameter. Primitives "send" and "receive" in figure 2.3 require explicit process addressing.

2. Implicit addressing: a process willing to communicate does not explicitly name a process for communication. Primitives "send_any" and "receive_any" in figure 2.3 support implicit process addressing. In the first primitive, the sender names a service instead of a process. According to Sinha (1997), this type of primitive is useful in client-server communications when the client is not concerned with which particular server, out of a set of servers providing the service desired by the client, actually services its request. On the other hand, in primitive receive_any, the receiver is willing to accept a message from any sender. This type of primitive is again useful in client-server communications when the server is meant to service requests of all clients that are authorized to use its service.

```
- Send a message to the process identified by process_id.
    send (process_id, message)

- Receive a message from the process identified by process_id.
    receive (process_id, message)

- Send a message to any process that provides the service of
  type service_id.
    send_any (service_id, message)

- Receive a message from any process and return the process
  identifier process_id of the process from which the message
  was received.
    receive_any (process_id, message)
```

Figure 2.3. Primitives for Explicit and Implicit Addressing of Processes

3.   Communication

The components of a distributed system are both logically and physically separated, however they must communicate in order to interact and perform tasks. We shall assume that all of the components that require or provide access to resources in distributed systems are implemented as processes. This is true for the client-server model outlined above. In client-server systems, a client process must interact with a server process whenever it requires access to a resource that it does not control.

As we saw previously, communication between processes involves operations in the sending and receiving processes that result in the transfer of data from the environment of the sending process to the environment of the receiving process, and the synchronization of the receiving activity with the sending activity, so that the sending or receiving process is blocked until the other process makes an action that frees it. To transfer data from one process to the other, the communicating processes must share a communication channel where synchronization is implicit in the operation of all programming primitives for communication.

The basic programming constructs take the form of programming primitives send and receive. Each message-passing action involves the transmission by the sending process of a set of data-values through a specified communication mechanism -- a channel or a port -- and the acceptance by the receiving process of the message. As

23

explained previously, the mechanism may be synchronous, or blocking, meaning that the sender waits after transmitting a message until the receiver has performed a receive operation and sent a reply; or it may be asynchronous or non-blocking meaning that the message is placed in a queue of messages waiting for the receiver to accept them and the sending process can continue execution almost immediately. Receive normally blocks the receiving process when no message in currently available.

The practical implementation of messages passing between processes located in different computers requires the use of a communication network for the transmission of data and for communication of synchronization signals. Distributed systems can be designed entirely in terms of message-passing, but there are certain patterns of communication that occur so frequently and are so useful that they can be regarded as an essential part of the support for the design and construction of distributed systems (Coulouris, Dollimore and Kindberg, 1996).

The two patterns of communication most commonly used are the client-server communication model for communication between pairs of processes and the group multicast communication model for communication between groups of cooperating processes. The performance of the communication subsystems used for interprocess communication is critical for the performance of distributed systems. High performance distributed systems require optimized implementations of these two patterns of communication.

a. Client-Server Communication Model

The client-server communication model is oriented towards service provision. Communication between client and server processes consists of the following:

1. Transmission of a request from a client process to a server process;
2. Execution of the request by the server;
3. Transmission of a reply to the client.

Figure 2.4 illustrates the client-server communication-model. The server process must become aware of the request message sent by the client process as soon as it arrives, and the activity issuing the request in the client process must be blocked, after the transmission of the message, until the reply from the server has been received.

24

```
                    Client                    Server

Send (message);
execution suspended          Request
                                        Receive (message);
                                        execution resumed


                                        Send (reply);
                                        execution suspended
Receive (reply)
Execution resumed           Reply
                                             ———  Blocked state
                                             -----  Executing state
```
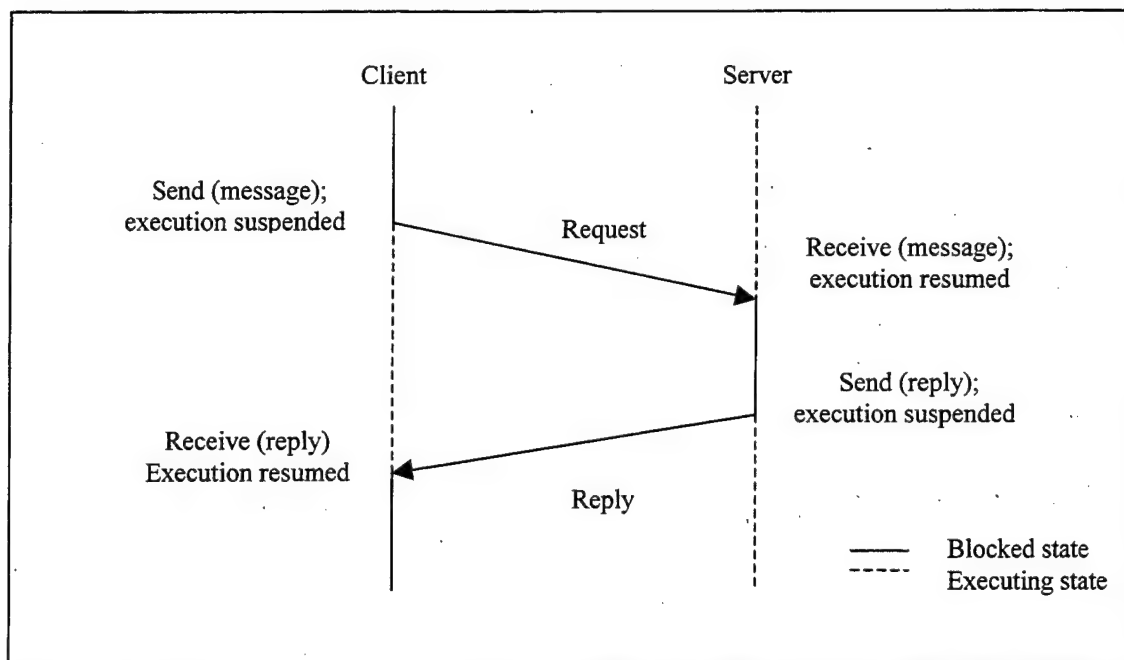
Figure 2.4. Client-Server Communication

According to Coulouris, Dollimore and Kindberg (1996), the client-server pattern of communication can be implemented in terms of the basic message-passing operations send and receive outlined above, but it is commonly presented at the language level as a remote procedure call (RPC) construct, which we will see in the next section. Note that a process is a client or a server only for purposes of a particular communication. A server can request the services of another server, and so can be a client of other processes. Similarly, a client can be a server to other processes.

b.   *Group Multicast Communication Model*

In the group multicast pattern of communication, processes interact by message passing, but in this case, the target of a message is not a single process but a group of processes. There are multiple receivers for a message sent by a single sender. Figure 2.5 illustrates this pattern of communication. Group multicast communication is useful for several practical applications, as shown by the following examples:

1. Locating an object: a client multicasts a message containing the name of a file directory to a group of file server processes. Only the one, which holds the relevant directory, replies to the request.
2. Fault tolerance: a client multicasts its request to a group of server processes all of which process the request identically. A group of two or more servers can continuously provide the service, even if one of their members crashes.

25
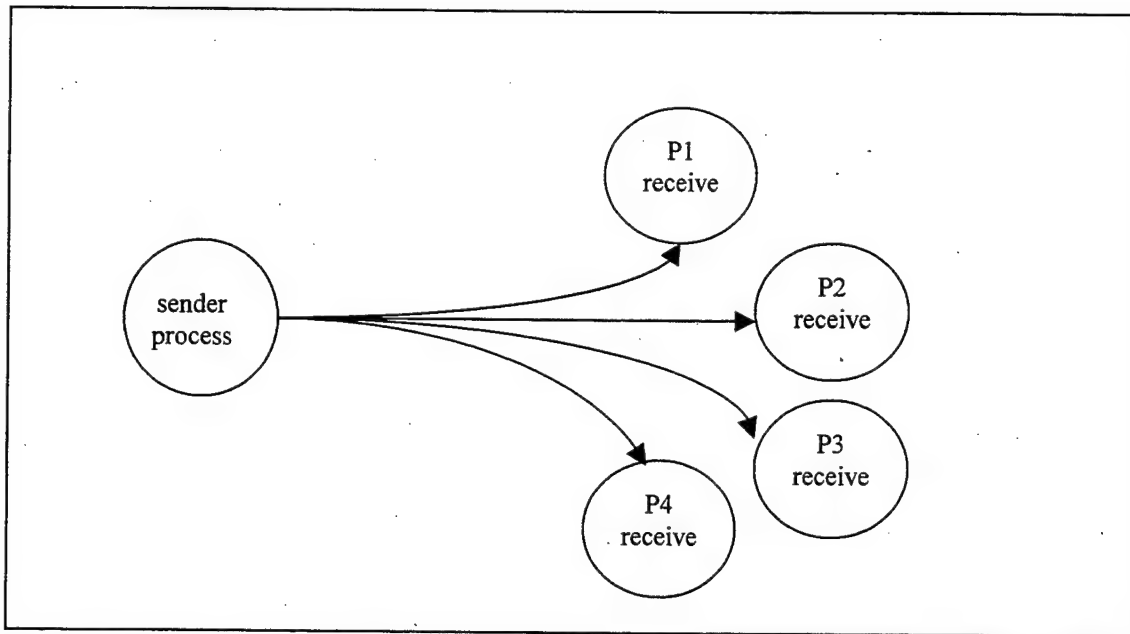
Figure 2.5. Multicasting to a Process Group

D.   REMOTE PROCEDURE CALL

The general message-passing model of interprocess communication (IPC) was presented in the previous section. The IPC part of a distributed application can often be adequately and efficiently handled by using an IPC protocol based on the message-passing model. However, an independently developed IPC protocol is tailored specifically to one application and does not provide a foundation on which to build a variety of distributed applications. Therefore, there is a need for a general IPC protocol that can be used for designing several distributed applications (Sinha, 1997).

Bierrel and Nelson (1984) introduced a different way to approach the client-server model. They suggested that programs should be allowed to call procedures located in other machines. When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B. Information can be transported from the client to the server process in the parameters and can come back in the procedure result. No message passing or I/O is visible to the programmer. This method is known as Remote Procedure Call (RPC).

Although this idea sounds simple, there are subtle problems. Because the calling and called procedures run on different machines, they execute in different address spaces, which causes complications. Parameters and results also have to be passed, which may be complicated, especially if the machines are not identical. Finally, both machines can

26

crash, and each of the possible failures causes different problems. Still, most of these can be dealt with, and RPC is a widely used technique that underlies many distributed operating systems (Tanenbaum, 1995).

In general, a service manages a set of resources on behalf of its clients. Clients can access them only by calling the procedures supplied by the service. The servers providing a service receive requests from the clients and execute the requested procedures.

At the RPC level, a service may be viewed as a module with an interface that exports a set of procedures appropriate for operating on some data abstraction as a resource. The ability to combine a group of procedures and variables in a module and to export only selected procedure names was introduced in programming languages such as Modula-2 and Ada as a method for structuring programs. The procedures exported by a module are generally defined to provide a complete set of operations on a given type of resource (Coulouris, Dollimore and Kindberg, 1996).

From the perspective of the client programs, a service provides the same facilities as software module, enabling clients to import its procedures. Normally, a server process runs indefinitely and its operations may be invoked by many clients. This enables the resources that it manages to be shared between clients.

1.   Remote Procedure Call Characteristics

The aim of a remote procedure calling mechanism is to maintain as far as possible the semantics of conventional procedure calls in an implementation environment that differs radically from that of conventional procedure calling. The main aspects of the semantics of the RPC are as follows:

1. The definition of a remote procedure specifies input and output parameters. Input parameters are passed to the server by sending values of the arguments in the request message and copying them into variables that are passed as parameters to a procedure in the server's execution environment. Output parameters are returned to the client in the reply message and they are used to replace the values of the corresponding variables in the calling environment.
2. Input parameters provide a direct correlation to parameters passed by value in conventional procedure calls. However, to implement parameter passing by reference, further information is needed, indicating whether each such parameter is used for input, output, or for both input and output. The need to specify these alternatives is one of the reasons why an interface definition language is an essential component of any RPC system.

27

3. A remote procedure is executed in a different execution environment from its caller and therefore can not access variables in the calling environment, such as the global variables declared by the caller.
4. It is meaningless for a process to pass addresses of memory locations or their equivalent in messages to other processes. Thus, the arguments and results of remote procedures can not include data structures that contain pointers to memory locations.

According to Coulouris, Dollimore and Kindberg (1996), the last restriction is less serious than it might appear to be. There is generally no need to transmit complex data structures in their entirety between servers and clients. If the need does arise to transmit data structures containing pointers, the structure must be "flattened" before they can be transmitted in messages. The process that manages a particular data structure is responsible for flattening and expanding it. Thus, a tree structured list might be flattened by converting it to a bracketed expression.

A service is accessed by means of calls to the remote procedures that it offers. Because of the differences between local and remote procedures and because a service should be defined at a level appropriate for the widest possible use, the RPC interface is not necessarily the most convenient for client programs. For this reason, and because there are some tasks that must be performed by a client (such as the location of a suitable server), the use of services by application programs is often supported by a user package. This package is a library of conventional procedures that presents a convenient procedural interface for use by application programs. The actual remote procedure call to servers is embedded within the user package.

2.   Design Issues

RPC systems fall into two classes. In the first class, the RPC mechanism is integrated with a particular programming language that includes a notation for defining interfaces. In the second class, a special-purpose interface definition language is used for describing the interfaces between clients and servers.

As an example, the Argus language developed by Liskov at MIT is designed for the construction of distributed programs. Remote procedure calls are integrated into the language (Liskov, 1988). Argus provides guardians: modules that are used to provide services and intended to be accessed by remote procedure call. The procedures in a

guardian are called handlers and a call to a handler is automatically treated as a remote call.

The second class includes Sun RPC, on which the Sun Network File System is based. The separate interface language approach has an advantage: it is not tied to a particular language environment. Although in practice, almost all examples of this approach are used in a C programming environment (Coulouris, Dollimore and Kindberg, 1996).

a. *Interface Definition Language*

An RPC interface definition specifies those characteristics of the procedures provided by a server that are visible to the server's clients. The characteristics that must be defined include the names of the procedures and the type of their parameters. Each parameter should also be defined as input, output, or in some cases both, to enable the RPC system to identify which values should be marshalled into the request and reply messages. An interface specifies a service name that is used by clients and servers to refer to the service that is offered by the collection of procedures.

b. *Exception Handling*

Any remote procedure call may fail when it cannot contact a server, whether the server is down or just busy. Therefore, remote procedure calls must be able to report error types that are due to distribution (such as time-outs), as well as, those that relate to procedure execution. Because any RPC may fail, an RPC system requires an effective exception handling mechanism for reporting such failures to the caller.

c. *Delivery Guarantees*

Request-reply protocols can be implemented in different ways to provide different delivery guarantees. The main choices are (Coulouris, Dollimore and Kindberg, 1996):

1. Retry request message: whether to transmit the request message until either a reply is received or the server is assumed to have failed;
2. Duplicate filtering: when retransmissions are used, whether to filter out duplicates at the server;
3. Retransmissions of replies: whether to keep a history of reply messages to enable lost replies to be retransmitted without re-executing the server operations.

The combination of these choices leads to a variety of possible semantics for the reliability of remote procedure calls as seen by the caller. The semantics are as follows:

1. Maybe call semantics: If for any reason a reply message has not been received after a time-out and there are no retries, the clients can not tell whether remote procedures have been called or not. This choice is not generally acceptable.
2. At-least-once call semantics: Retransmission of request messages without filtering of duplicates. In cases when the request message is retransmitted, the server may receive and execute it more than once. This choice may be acceptable only if a server can be designed with idempotent operations in all of its remote procedures.
3. At-most-once call semantics: Filtering of duplicates and retransmission of replies without re-executing operations. Some operations can have the wrong effect if they are performed more than once.

The at-most-once call semantics is the one usually chosen in RPC implementation. Bierrel and Nelson (1984) guarantee in Cedar RPC that if the server does not crash and the client receives the result of a call, then the procedure has been executed exactly once. Otherwise, an exception is reported and the procedure will have been called either once or not at all.

d. *Transparency*

Remote procedure calls should be as much like local procedure calls as possible. However, RPCs are more vulnerable to failure than local calls, since they involve a network, another computer and another process. Also, they consume much more time than local calls. Therefore, it can be argued that programs that make use of remote procedures must handle errors that cannot occur in local procedure calls.

The choice as to whether RPCs should be transparent is also available to the designers of interface languages. In the transparent case, the client calls remote procedures in the normal way for the language in use. In the non-transparent case, the client uses a special notation for calling remote procedures with the advantage that this notation may provide the ability to express requirements for distributed programming. For example, to specify a call semantics or to handle exceptions (Coulouris, Dollimore and Kindberg, 1996).

Liskov and Scheifler (1982) say that although the RPC system should hide low-level details of message passing from the user, the possibility of long delay or failure should not be hidden from the caller. The caller should be able to cope with failures

according to the demands of the application possibly by terminating an RPC, and in that case, it should have no effect.

3. Implementation

To achieve the goal of semantic transparency, the implementation of an RPC mechanism is based on the concept of stubs, which provide a perfectly normal (local) procedure call abstraction. It involves the transfer of arguments from the client process to the server process and the transfer of results from the server process to the client process. These arguments and results are basically language-level data structures, which are transferred in the form of message data between the two computers involved in the call.

The transfer of message data between the two computers requires encoding and decoding of the message data. The encoding process involves the conversion of program objects into a stream form that is suitable for transmission. This is known as marshalling. Unmarshalling is the reconstruction of program objects from the message data that was received in the stream form. According to Coulouris, Dollimore and Kindberg (1996), the software that supports remote procedure calling has three main tasks, as noted below.

a. Interface Processing

An interface definition may be used as a basis for constructing extra software components of the client and server programs that enable remote procedure calling. These components are illustrated on Figure 2.6. Both client and server assign the same unique procedure identifier to each procedure in the interface and the procedure identifier is included in request messages.

A RPC system will provide a means of building a complete client program by providing a stub procedure to stand in for each remote procedure that is called by the client program. The purpose of a client stub procedure is to convert a local procedure call to a remote procedure call to the server. The types of the arguments and results in the client stub must conform to those expected by the remote procedure. This is achieved by the use of a common interface definition. The task of a client stub procedure is to marshall the arguments and to pack them up with the procedure identifier into a message, send the message to the server, await the reply message, unmarshall it and return the results.

31
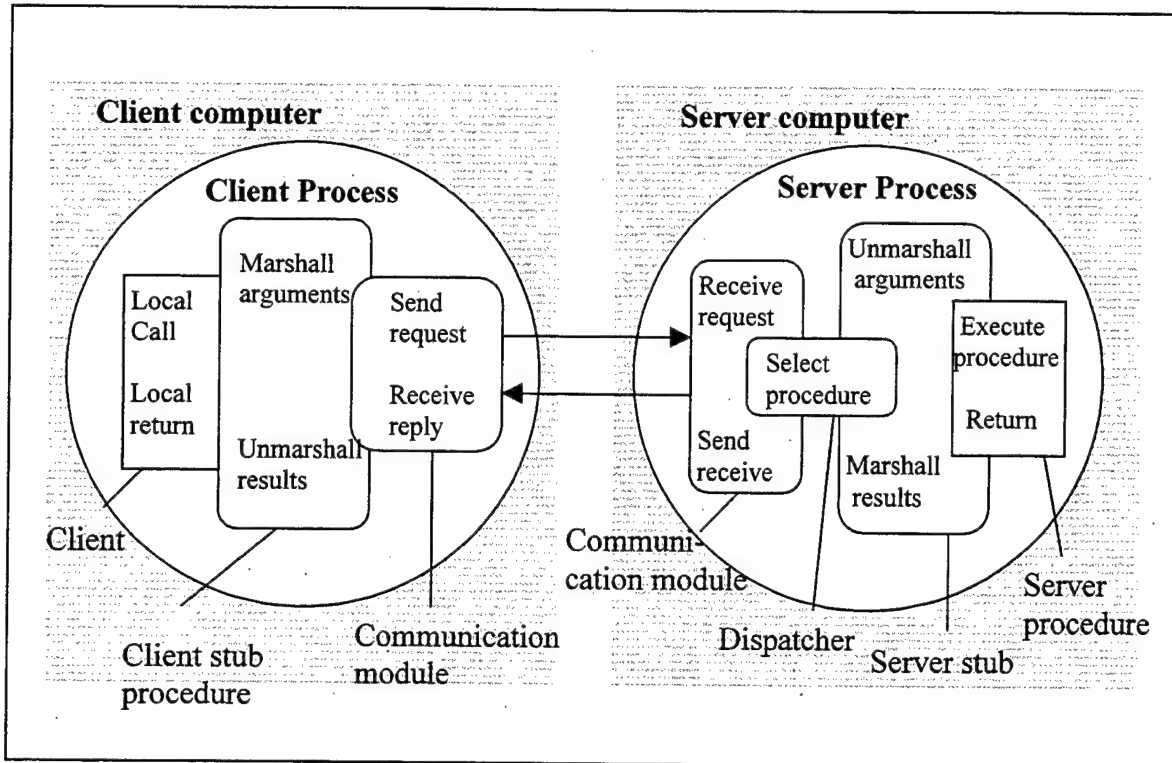
Figure 2.6. Stub Procedures (Coulouris, Dollimore and Kindberg, 1996)

To build the server program, the RPC system will provide a dispatcher and a set of server stub procedures. The dispatcher uses the procedure identifier in the request message to select one of the server stub procedures and pass on the arguments. The task of a server stub procedure is to unmarshall the arguments, call the appropriate service procedure, and, when it returns, to marshall the output arguments (or in case of failure an error report) into a reply message.

An interface compiler processes interface definitions written in an interface definition language. Interface compilers are designed to produce components that can be combined with client and server programs without making any changes to the existing compiler. An interface compiler normally performs the following tasks (Coulouris, Dollimore and Kindberg, 1996):

1. Generate a client stub procedure to correspond to each procedure signature in the interface.
2. Generate a server stub procedure to correspond to each procedure signature in the interface.
3. Use the signatures of the procedures in the interface to generate appropriate marshalling and unmarshalling operations in each stub procedure.

32

4. Generate procedure headings for each procedure in the service from the interface definition. The programmer of the service supplies the bodies of these procedures.

b. *Communication Handling*

The task of the communication handling module is to deal with communication between the client and server programs by using a form of request-reply communication, as described in the previous sections. The communication handling module is provided in forms suitable for linking with client and server programs.

c. *Binding*

An interface definition specifies a textual service for use by clients and servers to refer to a service. However, client request messages must be addressed to a server port.

Binding means specifying a map from a name to a particular object, usually identified by a communication identifier. The binding of a service name to the communication identifier of a specified server port is evaluated each time a client program is run. The form of communication identifier depends on the environment. For example, in a UNIX environment, it will be a socket address containing the internet address of a computer and a port number.

In a distributed system, a binder is a separate service that maintains a table containing maps from service names to server ports. A binder is intended to be used by servers to make their port identifiers known to potential clients, and by clients to obtain the addresses of the servers.

When a server process starts executing, it sends a message to the binder requesting it to *register* its service name and service port. If a server process terminates, it should send a message to the binder requesting it to *withdraw* its entry from the mappings. When a client process starts, it sends a message to the binder requesting it to *look up* the identifier of the server port of a named service. The client program sends all its request messages to this server port until the server fails to reply, at which point the client may contact the binder and attempt to get a new binding.

4. Asynchronous Remote Procedure Call

Remote procedure calls that do not receive replies are termed asynchronous. In an asynchronous RPC, the communication handler sends the request message and returns

33

control to the client program immediately, instead of blocking the client. In the synchronous case, the client marshalls the arguments, calls the *send* operation and then waits until the reply from the server arrives. Then it *receives,* unmarshalls and processes the results. After this, the client is able to continue its execution. In the asynchronous case, the client marshalls the arguments, calls the *send* operation and then immediately continues its execution. This arrangement allows the client and the server to work in parallel.

III. BACKGROUND KNOWLEDGE

A. REAL-TIME SYSTEMS MODEL

Real-time system is a technical term with a specific meaning. The requirements of real-time systems include timing constraints that must be met in the worst case for systems to be considered correct.

According to Luqi (1993), models for hard real-time systems are used to:

1. Support automated analysis to determine whether specified real-time constraints can be met by a given design for a given hardware configuration.
2. Help the designer construct a design that will meet a set of hard real-time constraints, if such a design exists.
3. Form a basis for the development of programming languages for hard real-time programming, for which it is possible to effectively determine whether a given program will always meet a given set of deadlines.

Models of real-time systems are essential for requirements analysis, specification and design of systems with real-time constraints. A coherent framework for classifying real-time constraints can be used to organize complex sets of timing requirements and to guide the process of discovering the timing requirements associated with an embedded software system.

There are a number of approaches to modeling real-time systems. This research is based on the model defined in the Prototype System Description Language (PSDL) (Luqi, Berzins and Yeh, 1988). PSDL is a language designed for clarifying the requirements of complex embedded systems, and for determining properties of proposed designs for such systems via prototype execution and static analysis. The language was designed to simplify the description of such systems and to support prototyping. PSDL is also the basis for a computer-aided prototyping system that speeds up the prototyping process by exploiting reusable software components and providing execution support for high level constructs appropriate for describing large real-time systems in terms of an appropriate set of abstractions (Luqi, 1993).

PSDL simplifies the design of systems with real-time constraints by presenting a high-level description in terms of networks of independent operators to the designer. The language provides simple and efficient synchronization and exception handling primitives.

35

PSDL has been designed both to ensure that the requirements are feasible and to provide the best service possible to the users of the proposed software. Also, PSDL provides a description of a proposed design that can be smoothly transformed into a final implementation after the requirements have been validated and the design has been verified.

1.   Real-Time Semantics of PSDL

PSDL is based on a computational model containing OPERATORS that communicate via DATA STREAMS, where each stream carries values of a fixed abstract data type (ADT). There are several ADTs already built into PSDL; the PSDL_EXCEPTION is one of them. Modularity is supported through the use of independent operators that can only gain access to other operators via data streams (Cordeiro, 1995).

The PSDL computational model treats software systems as networks of operators communicating via data streams. A PSDL decomposition is represented as an augmented directed graph (Luqi, Berzins and Yeh, 1988):

$G = (V, E, T(v), C(v))$, where:

- V is a set of vertices (vertices represent operators in the network);
- E is a set of edges (edges represent data streams in the network);
- $T(v)$ is the set of timing constraints for each operator $v \in V$;
- $C(v)$ is the set of control constraints for each operator $v \in V$.

The graph (V, E) in a PSDL decomposition determines the possible interaction between the operators. The timing and control constraints determine the conditions under which the operators are activated.

a.   Operators

Every PSDL operator is a state machine. Functional operators are machines with only a single state. When an operator fires, it reads one data value from each of its input streams, undergoes a state transition, and writes at most one data value into each of its output streams. The action of a PSDL operator is local, since its output values can depend only on the current set of input values and the current state of the operator. State transitions and input/output operations on data streams can occur only

when the operator fires. The firing of an operator can be triggered by the arrival of a specified subset of its input data values or by a periodic temporal event.

A PSDL operator can be either atomic or composite. Operators that are decomposed into lower levels are called composite operators, and they represent networks of components. This decomposition is always functional. An operator that is not decomposed is called atomic.

b. Data Streams

A data stream is a communication link connecting two sets of operators, the producers and the consumers. A PSDL data stream carries instances of an abstract data type associated with the stream, which can be a special pre-defined type representing exceptions. There are two different kinds of data streams: data flow streams and sampled streams.

Data flow streams are similar to FIFO queues with a length of one. Any value placed into the queue must be read by another operator before any other data value may be placed into the queue or it will overflow. Values read from the queue are removed from the queue and, if any attempt is made to read from an empty queue, it will underflow. Sampled streams represent a continuous source of data, of which the most recent value is meaningful. The most recently written value in a sampled stream must be available at all times and may be read many times or overwritten by more recent data before it is read. Some values may never be read, because they are replaced before the stream is sampled. In summary, it could be said that a data flow stream guarantees that none of the data values are lost or replicated, while a sampled stream does not make such a guarantee (Cordeiro, 1995).

c. State Streams

A State Stream can be either a data flow stream or a sampled stream, depending on the triggering condition of the consumer operator, but state streams must have been assigned an initial value for the stream.

An operator is a state machine if it has one or more state streams. The data flow diagram of a composite state machine operator has cycles, which represent the feedback loops that update the state variables. Every feedback loop must be broken by a

37

state stream. State streams must also be used when connecting time-critical and non time-critical operators.

d. Types

All PSDL data types are immutable, so that there can be no implicit communication by means of side effects. Both mutable types and global variables have been excluded from PSDL to help prevent coupling problems. The PSDL data types include the immutable subset of the built-in types of Ada user defined abstract types, the special types TIMER and EXCEPTION, and the types that can be built using the immutable type constructors of PSDL .

e. Exceptions

PSDL exceptions are values of a built-in abstract data type called EXCEPTION. This type has operations for creating an exception with a given name, for detecting whether a value is an exception with a given name, and for detecting whether a value is normal, which means that it belongs to some data type other than EXCEPTION. Values of type EXCEPTION can be transmitted along data streams just like values of the normal type associated with the stream.

f. Timers

Timers are software stopwatches that are used to record the length of time between events, or the length of time the system spends on a given state. This facility is needed to express relatively sophisticated aspects of real-time systems, such as timeouts and minimum refresh rates. They are governed by the PSDL control constraints START TIMER, STOP TIMER and RESET TIMER.

2. Control Constraints

The control abstractions of PSDL are represented as enhanced data flow diagrams augmented by a set of control constraints. The order of execution is only partially specified, and is determined from the data flow relations given in the enhanced data flow diagrams, but also affected by the types of data triggers among operators.

The control aspects of a PSDL operator are specified implicitly via control constraints, rather than giving an explicit control algorithm. There are several aspects to be specified, such as whether the operator is PERIODIC or SPORADIC, the triggering condition, and output guards.

a.   *Periodic and Sporadic Operators*

PSDL supports both periodic and sporadic operators. Periodic operators are triggered by the scheduler at approximately regular time intervals, so that they start execution somewhere after the beginning of each period, and complete by some deadline, which defaults to the end of the period. Sporadic operators are triggered by the arrival of new data values, possibly at irregular time intervals.

b.   *Data Triggers*

Any PSDL operator can have a data trigger. There are two types of data triggers in PSDL as illustrated by the following examples:

OPERATOR p TRIGGERED BY ALL x, y, z

OPERATOR q TRIGGERED BY SOME a, b

In the first example the operator p is ready to fire whenever new data values have arrived on all of the three streams x, y, and z, although there may be other streams arriving at the operator p, in which case the data values do not need to be new. The data streams associated with x, y, and z are data flow streams. This kind of trigger should be used when the items in a stream represent discrete events like, for example, transactions on a bank account, rather than samples from a continuous source of data (e.g., reading from a sensor). Also, it can be used to guarantee that the output of the operator is always based on fresh data for all the inputs in the triggering set.

The most important design consideration when " TRIGGERED BY ALL" is used is management of the firing frequencies of the producer and consumer operators. The period of the consumer operator must be smaller than or equal to the period of the producer, or stream buffer overflow errors will result. There is no problem if the period of the consumer operator is less than the period of the producer, since the actual firing rate of the two operators will be the same because data streams are tested for new information prior to the actual firing of the consumer.

In the second example, the operator q fires whenever new data arrives on at least one of the inputs a and b. This kind of activation condition guarantees that the output of the operator q is based on the most recent data value from at least one of its critical inputs a and b mentioned in the activation condition for q. This kind of trigger can be used to keep software estimates of sensor data up to date.

39

Every operator must have a period or a data trigger, or both. If a periodic operator has a data trigger, the operator is conditionally executed with the data trigger serving as input guard.

c. Execution Guards

The firing of a PSDL operator can be controlled by an execution guard. Execution guards provide conditional execution of operators based on conditional statements, which are evaluated prior to the firing of the associated operator. Execution guards can depend on data from any incoming data stream and they can be combined with the "TRIGGER BY ALL" and "TRIGGER BY SOME" data triggers mentioned above. Even if an execution guard is not satisfied, the data values are read and consumed from all the input streams, without firing the operator. Two examples of operation with triggering condition are shown below:

OPERATOR r TRIGGERED BY SOME x, y IF x = NORMAL AND y >10.0

OPERATOR s TRIGGERED IF x = NORMAL

d. Output Guards

Output guards do not affect the firing of an operator, which will fire regardless of whether or not its output is written to an output data stream. An output guard provides conditional transmission of computed results. An example is shown below:

OPERATOR t OUTPUT z IF z > 20 AND z < max

The example shows an operator with an output guard, which depends on the input value max and the output value z. The condition of an output guard may depend on the output values of the operator, on the values read from the input streams, and on values of timers.

e. Exception Guards

Exception guards provide conditional raising of exceptions. Exceptions are transmitted on all of the output streams of the operator that raised the exception. For example:

OPERATOR f EXCEPTION e IF x > 50

This control constraint transmits the exception value named e on all of the output streams of f that are of type EXCEPTION instead of the values actually computed

by f whenever the input value x is greater than 50. Exception guards can be used to check assumptions about validity of inputs, outputs, or states.

3. Timing Constraints

PSDL operators can be subjected to timing constraints, which are specified by giving bounds on the duration of various kinds of time intervals. Operators can be time-critical and non time-critical, depending on whether or not they are assigned a value for the maximum execution time (MET) by designer. If an operator is time-critical, it can be further subdivided into periodic or sporadic operator subtypes. Periodic operators are explicitly assigned a frequency (PERIOD) of execution, meaning that they will fire within regular periods, exactly once, but not necessarily at regular intervals of time. Sporadic operators are not explicitly assigned a period, but they fire whenever the data trigger is satisfied. However, sporadic operators have a minimum interval of time between successive firings (Cordeiro, 1995).

Timing constraints are an essential part of specifying real-time systems. Table 3.1 summarizes the timing constraints supported by PSDL.

Constraint	Symbol	Applies to	Constrains	Default
Max. execution time	MET	Time critical op.	CPU time	-
Period	PER	Periodic op.	Activation, next activation	-
Finish within	FW	Periodic op.	Activation, completion	PER
Max. response time	MRT	Sporadic op.	Activation, completion	Heuristic
Min. calling period	MCP	Sporadic op.	Activation, next activation	MRT-MET
Latency	L	Streams	Write, next read	0
Min. period	MP	Streams	Write, next write	0

Table 3.1. PSDL Timing Constraints

The MET is the maximum amount of serial CPU time required to execute an operator under worst-case conditions. Note that for atomic operators the MET complies with the above definition. For the composite operator, however, the MET is the maximum CPU time needed along any single thread of control. The static scheduler must ensure that at least this much CPU time is allocated to an operator between each activation time and its deadline. This CPU time need not be all in one contiguous interval and it need not be all on the same processor, in which case any inter-processor communication delays must be considered.

When decomposing a complex operator into a network of simpler operators, the designer of a real-time system must consider timing requirements as well as functional requirements. The time available for the execution of the components of a composite time-critical operator is constrained by different necessary conditions for feasibility, depending on the type of target hardware for the proposed system. For the uniprocessor case, the sum of the maximum execution times of the components in a proposed decomposition must be less than or equal to the MET of the composite operator. Whereas for the multiprocessor case, the sum of the maximum execution times of the components along every path from an external input to an external output in a proposed decomposition must be less than or equal to the MET of the composite operator.

The MRT defines an upper bound on the time between the arrival of new data that satisfies the data triggering condition of a sporadic operator and the time when the last value is written onto the output stream. The MRT applies only to sporadic operators (Cordeiro, 1995).

The MCP also applies to sporadic operators and represents a lower bound on the time between two consecutive triggerings of a sporadic operator. This lower bound defines the maximum input rate under which the sporadic operator must meet its deadline. Thus, the MCP characterizes the worst-case operating conditions under which a proposed design is guaranteed to meet its real-time constraints. The scheduling delay for a sporadic operator is the interval of time between the satisfaction of consumer operator's triggering conditions and the corresponding reading of the input values by the consumer. In general, a sporadic operator is triggered by new data values on a subset of the input streams. The sporadic operator is activated when the last new values of the data trigger set become available for reading by the operator. On distributed architectures, there may be some intervening communication delay if the operator that wrote the value is allocated to a different processor than the sporadic operator that reads the value (Luqi, 1993).

The timing constraints for a sporadic operator are illustrated in Figure 3.1. As we will see later, sporadic events are handled by polling in CAPS, where sporadic operators are converted into equivalent periodic ones, whose period is called the triggering period (TP).
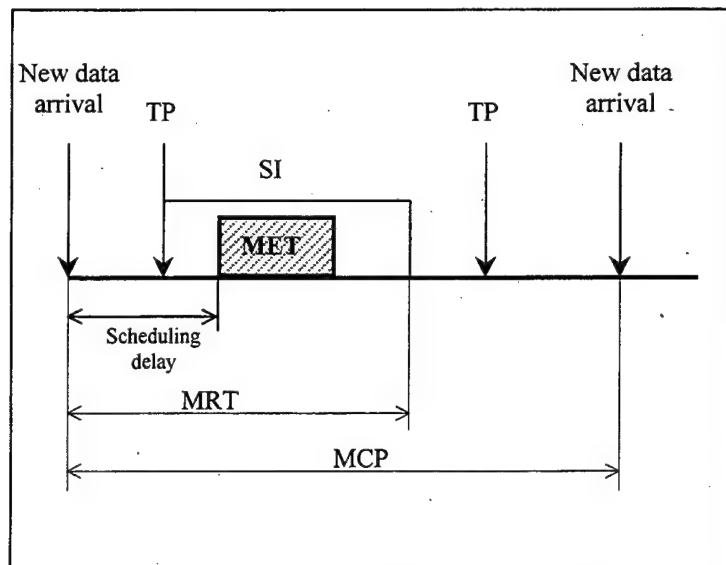
Figure 3.1. Sporadic Timing Constraints

Periodic operators are triggered by temporal events, which must occur at regular intervals. For each operator, these activation times are determined by the specified period (PER), which is the time interval between two successive activations. The differences between sporadic operators and periodic operators are that the period of a periodic operator is fixed and the activation events are defined based on the absolute time, instead of the arrival of input data. Note, however, that there is a distinction between activation time and the actual start time of a periodic operator as shown in Figure 3.2.

Finish within (FW) defines an upper bound on the time interval between each activation and completion of firing. The difference between the activation time and its deadline is called the scheduling interval (SI) and it is equal to FW.
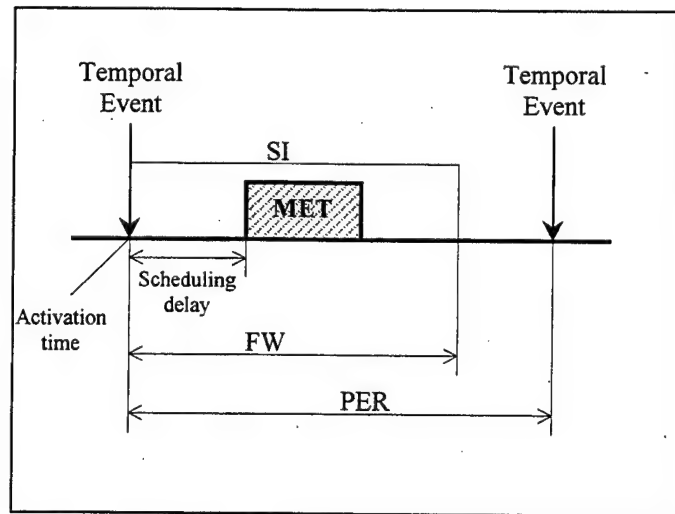
Figure 3.2. Periodic Time Constraints

Scheduling intervals of a periodic operator can be viewed as a fixed window of a size equal to FW, evenly separated by the period PER, and whose absolute position on the time axis is determined by the start time t of its first execution. For the first instance this time may vary within the closed interval [0, PER] of the operator, and is called the phase of the operator. Scheduling intervals for sporadic operators will be discussed in the next section.

To express the behavior of distributed systems, PSDL provides two timing constraints, Latency (LAT) and the Minimum Output Period (MOP). The latency of a stream is an upper bound on the time from when a data value is written into the stream to the time when the data value can be read from the stream. Latency models networks or telecommunication links, it specifies an upper bound on the allowable time spent by a stream in the network. This information should be used by the scheduler to simulate the worst case behavior for the delay in the network.

The minimum output period is a lower bound on the duration of the interval between two successive write events on the stream. The purpose of the Latency and MOP constraints is to declare communication constraints that arise from hardware limitations imposed by external constraints on how the software functions must be allocated to different physical nodes of a distributed system. The effect of these constraints on static scheduling is that data can not be read from a stream until a delay

44

equal to the latency has elapsed, and data can not be written into a stream until the minimum period has elapsed.

4. Synchronization in PSDL

As we saw above, there are two kinds of streams in PSDL: sampled streams and data flow streams. However, within sampled streams there are two semantically different subtypes of stream, depending on the triggering condition of the consumer operator. If the consumer operator is not triggered by any data, then specific data value can be lost or overwritten, or even read over and over again by the consumer, without any problem. According to Cordeiro (1995), this type of behavior is very useful when reading sensor data. In most cases, the sensor will be able to generate data in much higher rate than the consumer will read it, but the most recent data is of primary interest.

The second type of sampled stream exists when the consumer operator is TRIGGERED BY SOME data value. In this case, the consumer should always read the new data from one of the streams specified in the TRIGGERED BY SOME clause. Although buffer overflow or underflow is not an issue, due to the way sample streams are defined, the only way to avoid loss of data in this case is to enforce the condition that $PER_{producer} \geq PER_{consumer}$, and, consequently, the synchronization problem will have to be handled accordingly (Cordeiro, 1995).

In the case of data flow streams, the consumer is TRIGGERED BY ALL. The streams specified in the TRIGGERED BY ALL clause should be examined, and if all of them have new data in their buffer, they should be consumed, firing the consumer operator. The TRIGGERED BY ALL condition can be thought of as being a logical AND among streams declared in this clause. Again, in this case, there is a need to enforce $PER_{producer} \geq PER_{consumer}$ so that no data is lost, and once again the synchronization problem must be handled explicitly. The basic semantic difference between the TRIGGERED BY ALL data flow streams and the TRIGGERED BY SOME sampled streams is that if the data is not consumed and a new data arrives, in the former it will raise a buffer overflow exception, while in the latter the data will be simply overwritten.

45

5. Mutual Exclusion

Since updates to state variables must be serialized to preserve the integrity of the data, any pair of operators that belong to a common cycle in the expanded data flow graph must not be scheduled concurrently. If two such operators are allocated to different processors of a distributed system, then the scheduler must provide sufficient time between the completion of one such operator and the start time of the next to account for possible interprocessor communication delays (Luqi and Shing, 1996).

6. Hardware Models

The semantics of PSDL are independent of the hardware model, but scheduling and the feasibility of realizing the declared real-time constraints depend on the architecture and characteristics of the hardware system on which the proposed system will run (Luqi and Shing, 1996). The hardware models associated with PSDL can be characterized by the number of processors N, a vector of processor speed S_i, a matrix of interprocessor delays $D_{i,j}$ and a matrix of inverse link speeds (seconds per bits) $T_{i,j}$, where $D_{i,i} = 0$, $T_{i,i} = 0$, and $1 \leq i, j \leq N$.

According to the target architecture, we could have the following special cases:

1. Single processor: $N = 1$,
2. Identical processors: $S_i = s$,
3. Unlimited bandwidth: $T_{i,j} = 0$, and
4. Homogeneous network: $D_{i,j} = d$ for $j \neq i$.

For distributed systems we could derive the latency for the transmission of a data value b bits long from processor i to processor j as following:

$$L = D_{i,j} + (b \times T_{i,j})$$

B. REAL-TIME SCHEDULING ANALYSIS

According to Baker (1974), scheduling is the allocation of resources over time to perform a collection of tasks. This, rather than a general definition conveys the basic idea of scheduling theory, which is a collection of principles, models, techniques and logical conclusions that provide insight into the scheduling function.

This section is adapted from Cordeiro (1995) and is included here for the convenience of the reader. To have a better understanding of the context in which scheduling issues are found we present in Figure 3.3 a proposed taxonomy for the

46

scheduling function. This taxonomy is an enhancement done by Cordeiro (1995) of that proposed by Cheng, Stankovic and Ramamritham (1987).

As shown in the figure, classical scheduling and real-time scheduling are distinct areas. Most of the problem areas in classical scheduling make use of objective functions, such as minimizing flowtime, minimizing mean tardiness, and minimizing completion time (makespan), which does not convey much of the important information needed by real-time systems. Nevertheless, some of these results can provide very fruitful insights into real-time scheduling problems.
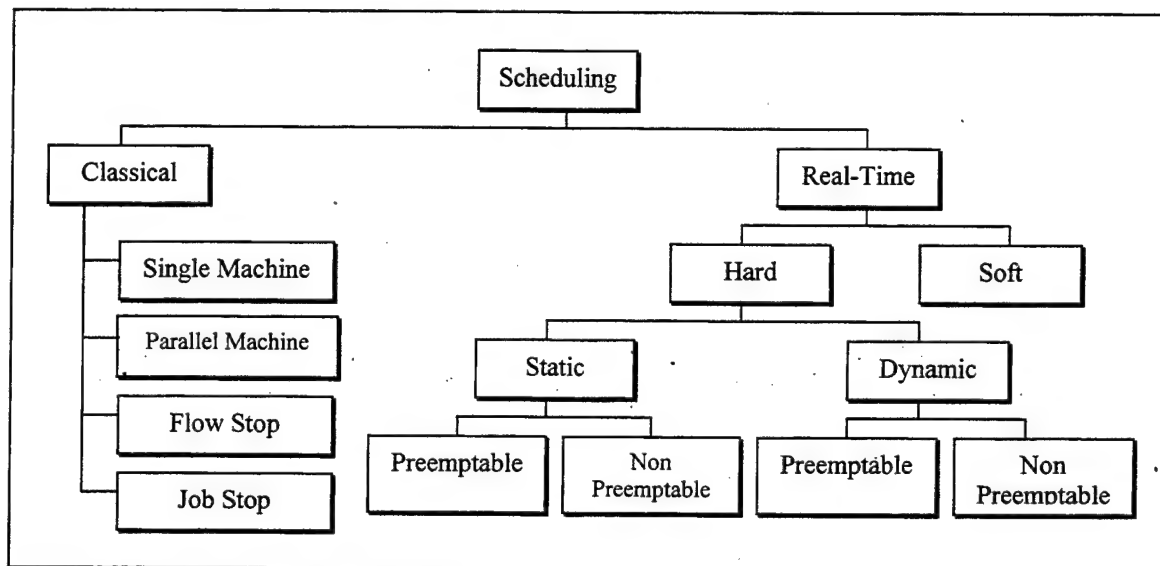
Figure 3.3. Scheduling Taxonomy

1. Real-Time Scheduling Model

An instance of a prototype T can be thought of as union of three disjoint finite sets, namely the set P of periodic operators, the set S of sporadic operators, and the set N of non-time critical operators. Each periodic operator o_i is completely specified by the tuple (MET, PER, FW) where MET is the maximum execution time used by each instance of operator o_i, PER is the period, and FW is the length of its scheduling interval. Likewise, each sporadic operator is completely specified by the tuple (MET, MCP, MRT)^{SP}, where MCP is the minimum period between two consecutive instances of the operator o_i, and MRT is the upper bound on the time between the triggering of the operator o_i by new data arrival in the data trigger set, and the completion of writing to all of its output streams. The superscript SP is used in the sporadic case, only to distinguish from the tuple of the periodic operator. Given any static schedule for a prototype T, we

shall use $st(o_{i,k})$, $ct(o_{i,k})$ and $d(o_{i,k})$ to denote the actual starting time, completion time and deadline of the kth instance of operator o_i in the schedule. In any feasible schedule, we must have

$$0 \leq st(o_{i,1}) \leq PER(o_i), \; d(o_{i,1}) = PER(o_i) + MET(o_i),$$

$$\text{activation\_time}(o_{i,k}) = st(o_{i,1}) + (k - 1) \times PER(o_i)$$

and

$$d(o_{i,k}) = \text{activation\_time}(o_{i,k}) + FW(o_i) \text{ for } k > 1$$

for every periodic operator o_i, where $st(o_{i,1})$ is called the phase of operator o_i as defined in the previous section.

By definition, every periodic operator must start and finish execution within its period of activation. In addition, the following restriction is imposed on the model:

$$MET \leq FW \leq PER$$

Clearly, this inequality is needed; otherwise, there is no way to execute such an operator within the specified amount of time (FW). Note that there may be a case where PER < MET; such processor demand can only be satisfied using pipelining in a multiprocessor environment (Luqi, 1993, and Luqi, Shing, and Brockett, 1993).

For the sporadic operator, all of the above assumptions are also applicable, since they will be converted into equivalent periodic operators, but the following restriction is imposed to the model:

$$MET \leq MRT, \; MET \leq MCP$$

Sporadic operators are activated by the arrival of new data on the input streams specified in the operator's control constraints. The activation time is the earliest time the triggering data is available for reading by the operator. For producer and consumer operators running on different processors, this is the time data is written by the producer plus the interprocessor communication delay, known as latency. Scheduling for sporadic operators is based on the following constraints.

$$\text{deadline}(o_{i,k}) = \text{actvation\_time}(o_{i,k}) + MRT(o_i)$$

$$\text{activation\_time}(o_{i,k+1}) \geq \text{activation\_time}(o_{i,k}) + MCP(o_i)$$

A prototype T is said to be schedulable if there exists a schedule such that the completion time for the execution of instance k of operator o_i, $ct(o_{i,k})$, is less than or equal to its corresponding deadline $d(o_{i,k})$, for all k and o_i, and the precedence constraints of the

prototype is satisfied. The precedence constraint between operator o_i and o_j, is written $o_i < o_j$, where $<$ denotes a partial ordering on the execution of tasks o_i and o_j, and is satisfied if

$$\forall \text{ instances } o_{i,k}, o_{j,k} \qquad (k - 1) \times PER(o_i) + st(o_{i,1}) < (k - 1) \times PER(o_j) + st(o_{j,1})$$

$$\text{and}$$

$$(k - 1) \times PER(o_j) + st(o_{j,1}) + \Delta < k \times PER(o_i) + st(o_{i,1})$$

where $(k - 1) \times PER(o_i) = (k - 1) \times PER(o_j)$ and Δ equals the maximum time to read input by operator o_j (Cordeiro, 1995).

Both periodic and sporadic operators are non-preemptable, which means that once they start execution they will run to completion. The non time-critical operators can be preempted. No idle time is inserted into the static schedule, unless there are no operators ready to execute. All timing information is assumed to be an integral multiple of a basic unit of time.

2. Non-Preemptive Tasks Scheduling

The purpose of this subsection is to present a series of theorems on schedulability for a set of independent, non-preemptive, periodic task sets. The objective is to provide the necessary background to build a framework upon which the later sections in this chapter will be based.

These theorems will help us to better understand the conditions for schedulability of non-preemptive tasks. For proofs of the theorems and further reading the reader is directed to the work of Cordeiro (1995).

a. *The Maximum Execution Time Theorem*

Theorem 1:

"For an independent periodic task set P, if $\exists$ some tasks X and Y $\in$ P, such that $MET_x \geq PER_y$ then P is not schedulable in the uniprocessor case by any non-preemptive algorithm. Furthermore, if X = Y then neither the preemptive nor the non-preemptive algorithms can find a feasible schedule." (Cordeiro, 1995)

This theorem is also valid for a sporadic task set when $MET_x > MCP_y$ for X = Y, the trivial case. However, for X $\neq$ Y the situation is slightly more complex and there are two cases to consider. The first is when $MRT_y < MCP_y$; this is clearly not

schedulable. The second case is $MRT_y \geq MCP_y$; the set is not schedulable if $MET_x + MET_y > MRT_y$. Figure 3.4 illustrates this case.
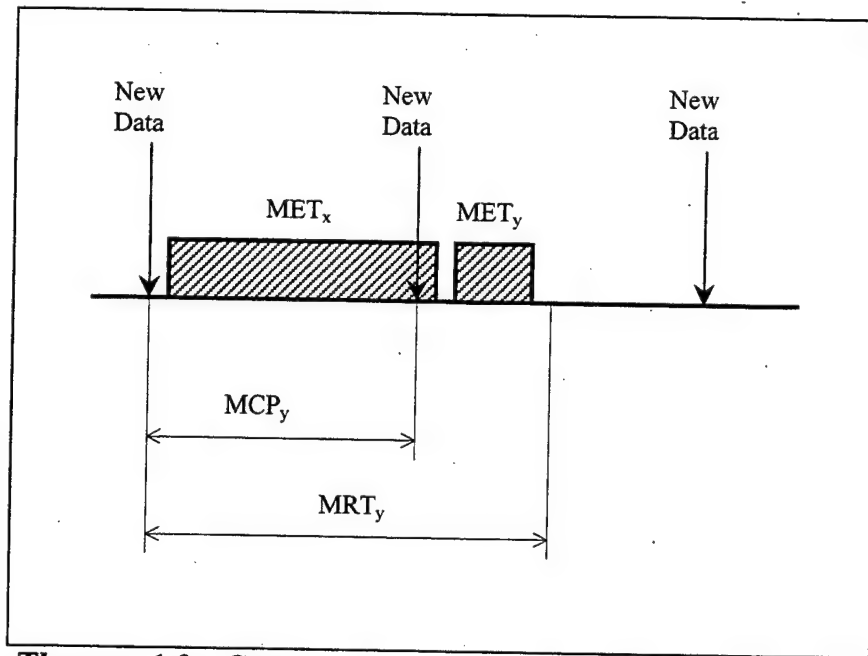
Figure 3.4. Theorem 1 for Case 2 of the Sporadic Task Set

Corollary (for the distributed case):

"For an independent periodic task set P, if $\exists$ some tasks X and Y $\in$ P, such that $MET_x \geq PER_y$, then in order for P to be schedulable in the multiprocessor case, tasks X and Y must be placed in different processors, and if X = Y, then it must be pipelined." (Cordeiro, 1995)

The conditions imposed on a task X for it to be pipelineable, as well as, detailed description of pipelining in this context can be found in the work of Luqi (1993) and Luqi, Shing and Brockett (1993).

b.   *The Finish-Within Theorem*

Theorem 2:

"For an independent periodic task set P if $\exists$ some indivisible task X $\in$ P such that $MET_x > FW_x$ then P is not schedulable under any scheduling algorithm, not even in a multiprocessor environment." (Cordeiro, 1995)

Note that this theorem can be extended to cover the sporadic case when $MET_x > MRT_x$. It is also applicable to the case where we have precedence constraints in the set P.

c. *The Minimum Period Theorems*

Theorem 3:

"For a periodic task set P, if $\forall$ tasks $X \in P$, $FW_x \geq PER_x$ and $\sum\limits_{x=1}^{n} MET_x \leq$ PER$_z$ where PER$_z$ denotes the minimum period in P, then P is schedulable." (Cordeiro, 1995)

The minimum period is certainly a divisor of the least common multiple of the periods (LCM), and, as such, it can span the entire LCM within an integral number of steps. It is a kind of sliding bin-packing where a sliding window of size equal to the minimum period is present and always large enough to fit all tasks present in that window. Depending on the periods, not all instances may be active simultaneously in that specific window. Figure 3.5 below shows the minimum period sliding window for a set of four tasks with the following timing constraints: task1 (-, 300, -), task2 (-, 200, -), task3 (-, 400, -) and task4 (-, 600, -). This theorem is valid even when precedence constraints are taken into consideration.
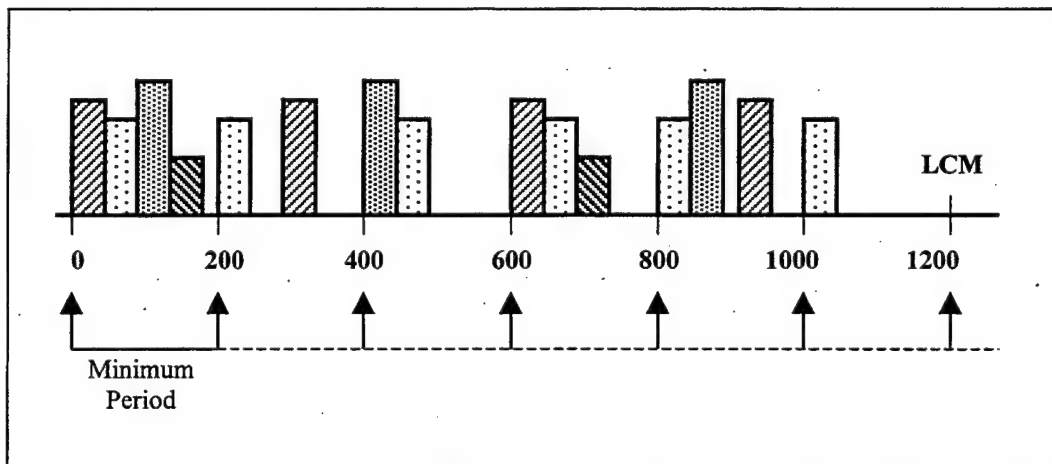
Figure 3.5. The Minimum Period Sliding Window

Theorem 4:

"For a periodic task set P, if $\forall$ tasks $X \in P$, $\sum\limits_{x=1}^{n} MET_x \leq FW_z$, where FW_z denotes the minimum FW in P, then P is schedulable." (Cordeiro, 1995)

The same idea of sliding bin-packing applies here. Now, however, the size of the bin must be decreased. The "bin" now should be understood to be the least value among all periods and FW from the tasks in P.

51

d. *The Load Factor Theorem*

Theorem 5:

"For a periodic task set P, if $\sum_{x=1}^{n} \frac{MET_x}{PER_x} > k$, where k is the number of available processors, then the set is not schedulable." (Liu and Layland, 1973)

Theorem 5 defines a necessary condition for the schedulability of a periodic task set, and it basically stipulates that if the summation of all individual load factors (MET_x/PER_x) is bigger than the number of available processors, then the set is not schedulable (Liu and Layland, 1973). We should note that theorem 5 is valid to both preemptive and non-preemptive algorithms (Zhu, Lewis and Colin, 1994).

e. *The Harmonic Block Theorem*

Theorem 8:

"If ∃ an infinite feasible schedule S without any inserted idle time for a periodic task set P with precedence constraints, such that the first instance of every task, T_j in P must start by time PER_j, then there exists an infinite feasible schedule S' consisting of a transient portion of length at most LCM, followed by a cyclic portion of length LCM that repeats forever." (Cordeiro, 1995)

The Harmonic Block (HB) is the least common multiple (LCM) of the periods of a periodic task set, as illustrated in Figure 3.6. It is the interval upon which the task set will be tested for schedulability. If a feasible schedule can be found within 2 × HB and if latencies are not allowed in the schedule, then it is possible to say that the same pattern can be repeated forever.

$$OP_1(190, 600, 600) < OP_2(20, 200, 200)$$
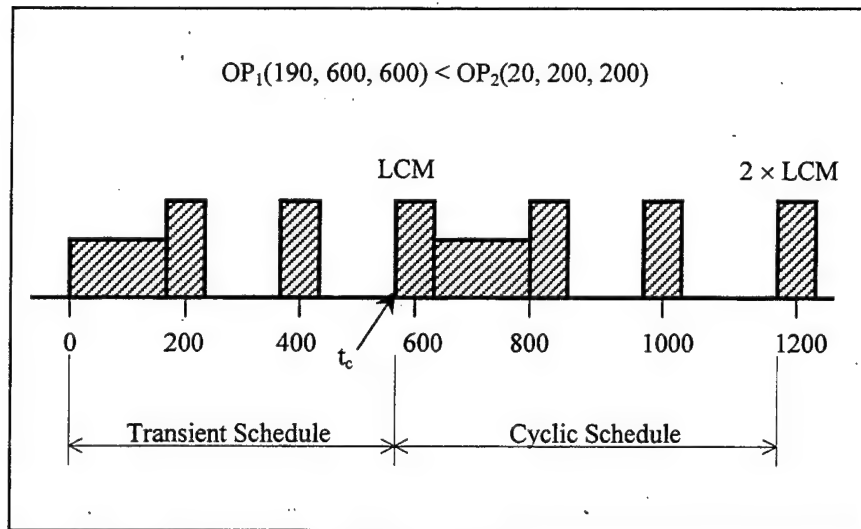
Figure 3.6. The Transient and Cyclic Schedules

It is a well known and accepted result that the LCM of the periods of a periodic task set provides a finite interval of time for which a cyclic schedule can be calculated, if one exists, and repeated forever (Mok, 1983). As we can see in Figure 3.6, it is not necessary for all task instances to start in the interval [0,LCM] and complete execution by time LCM for a cyclic feasible schedule to exist. It seems very reasonable to allow the first instance of a periodic task to start within its period of activation, but finish up to the end of the period plus its computational time. Actually, this would be very desirable if it could somehow improve the already difficult problem of non-preemptive scheduling.

3. Coping with Sporadic Tasks

Generally speaking, a sporadic task is defined as an aperiodic task that has a minimum duration between two consecutive activations. If that was not so, neither the static nor the dynamic approach could be used to guarantee schedulability.

If interrupts are used to detect the occurrence of aperiodic events at run-time, then a dynamic approach should be used. However, in the static scheduling framework, where all the task requests must be known *a priori* so that a fixed and static schedule can be generated, the only way to handle aperiodic, unanticipated sporadic task is to use a periodic process that functions as a polling device. Its main role is to check for arrival of sporadic tasks and to serve them during its allocated time slot. However, due to the random nature of aperiodic processes, we may not be able to handle a concentrated set of arrivals or even worse, not catch them all with the sporadic server approach. To

53

overcome this difficulty, several bandwidth preserving algorithms have been proposed. Among them could be mentioned the Priority Exchange, Deferrable Server and the Sporadic Server (Audsley and Burns, 1993).

In short, we need to guarantee that all time-critical sporadic operators will be serviced in a timely fashion, even in the worst case situation. The solution, considering the model we have been studying, is to use polling and assign one sporadic server for each time critical sporadic operator.

The next step is to convert the sporadic operator into a periodic one so that all the original timing constraints from the sporadic operator are still satisfied. Each sporadic operator will be assigned a triggering period (TP). The term triggering period will be used for the period of the converted sporadic operator and the usual term FW for its finish within. As shown in Figures 3.7 and 3.8, two cases can occur.

The first case is when $MCP < MRT - MET$ and the equivalent periodic operator must have $TP \leq MCP$ to satisfy the original time constraints. Also, it must enforce $FW = MRT - TP$, so that in the critical case (shown in Figure 3.7) the data that was missed by the previous triggering period can be consumed by the next TP and still finish within the original MRT.
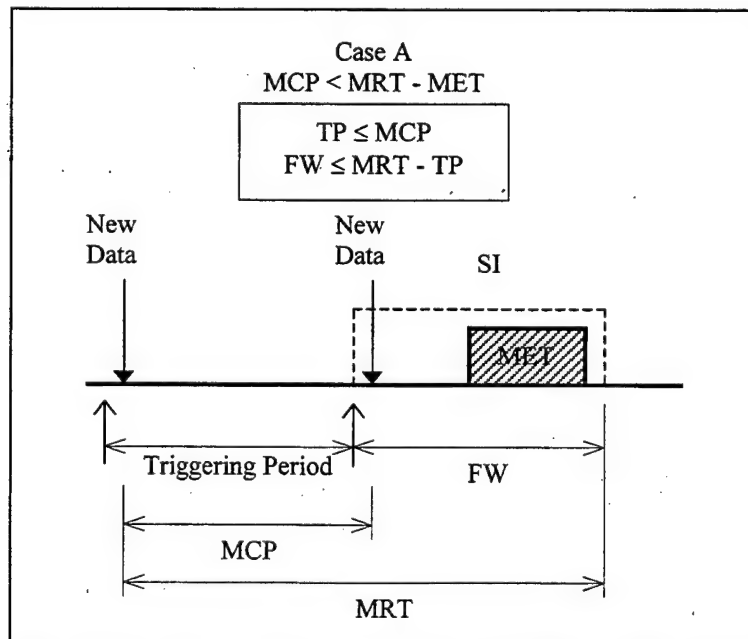
Figure 3.7. The Sporadic Conversion when MCP < MRT – MET
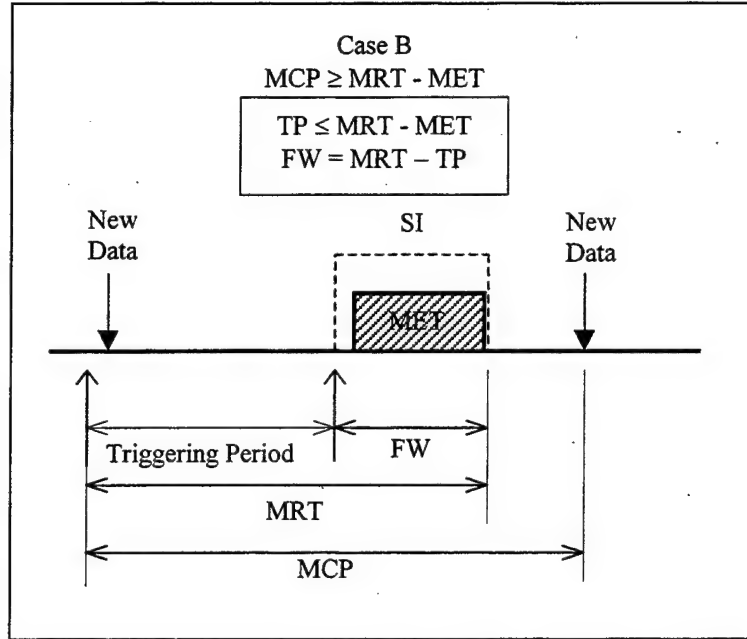
Figure 3.8. The Sporadic Conversion when MCP ≥ MRT - MET

The second case, shown in Figure 3.8, occurs when MRT - MET ≤ MCP. This more constrained situation forces a further reduction in the triggering period. Thus, the new TP should be TP ≤ MRT - MET and the FW should be equal to MRT - TP. In general, the triggering period should be

$$\text{MET} \le \text{TP} \le \min(\text{MRT - MET, MCP})$$

and

$$\text{FW} = \min(\text{TP, MRT - TP}).$$

Nevertheless, in order to minimize the impact on the load factor of the prototype, it is desirable that the TP be as large as possible, meaning that

$$\text{TP} = \min(\text{MRT - MET, MCP}).$$

Also, the MRT for a sporadic operator must be upper bounded by twice its MCP and lower bounded by twice its MET, as follows.

$$2 \times \text{MET} \le \text{MRT} \le 2 \times \text{MCP}$$

The worst case situation is when MRT assumes its lowest possible value, which is $2 \times$ MET. The triggering period TP will then reflect its lowest possible value, which is MET, with FW still being equal to MET.

According to Cordeiro (1995), when implementing this conversion it is strongly recommended that a careful analysis of the task graph be made to determine reasonable

bounds for the period of the transformed sporadic operator. At first glance, an obvious upper bound is the value of its MCP. However, for lower-bounds this choice is not so clear. Nonetheless, it is assumed that after this pre-processing there will be an interval of possible values for the period of the transformed sporadic task. These bounds provide us with some margin for making the conversion so that the final harmonic block of the entire set does not increase significantly.

C.   DISTRIBUTED SCHEDULING

In general, two different approaches to handling distributed computation can be identified. In the first, the distributed system is coordinated by a single system clock, which synchronizes all tasks so that computation progresses in a lock-step fashion, and communication between tasks can only occur at specific times. In the second approach, tasks are synchronized only when necessary, and do so by executing appropriate hand-shake protocols. The former approach requires less inter-processor communication, but is rigid, and relies on a global clock whose implementation is another very difficult problem to solve. Although more flexible, the latter approach dramatically increases the complexity of the synchronization problem and may be very costly in terms of communication, since many acknowledge signals must be exchanged in order to maintain proper synchronization. The use of rigorous and more constrained time requirements allows for the establishment of a weak form of synchronization among the tasks of the distributed system and represents an alternative in the middle (Mok, 1983).

According to Cordeiro (1995), the ideal real-time distributed system should have the following goals:

1. Be able to support groups of tasks running asynchronously in different processors;
2. Eliminate the need for enforcement of any kind of synchronization required by communication; and
3. All the deadlines and other requirements (such as loss of data, etc.) should be met.

A good alternative to achieve the ideal system is investigating ways of restricting or relaxing the timing requirements to increase the chances of finding a feasible schedule. In other words, the objective would be to change timing constraints so that no synchronization would be needed, and consequently decrease substantially the complexity of the distributed scheduling problem.

1.   Dealing with the Synchronization Problem

When dealing with the synchronization problem, Cordeiro (1995) studied the synchronization in PSDL and addressed several issues based on the real-time semantics of PSDL. He evaluated the different triggering conditions, stream types and operator types for PSDL graphs with no cycles, and his conclusion was that we need to enforce the condition that $PER_{producer} \geq PER_{consumer}$ so that no data is lost and, consequently, the synchronization problem will have to be handled accordingly. Also, he showed that even in the uniprocessor case, with the period of the consumer being smaller than the period of the producer (see Figure 3.9), synchronization is not always a good alternative. Figure 3.9 shows an example where no feasible schedule exists if synchronization is enforced, but one does exist otherwise. Three outcomes are possible if the synchronization is not required:

1. If the consumer operator is TRIGGERED BY ALL x,y , the proposed schedule is valid but x and y will be consumed one instance later,
2. If the consumer operator is TRIGGERED BY SOME x, y , then the schedule is always valid, because x and y do not need to be consumed together,
3. If there is no trigger, then the relative order is not important.
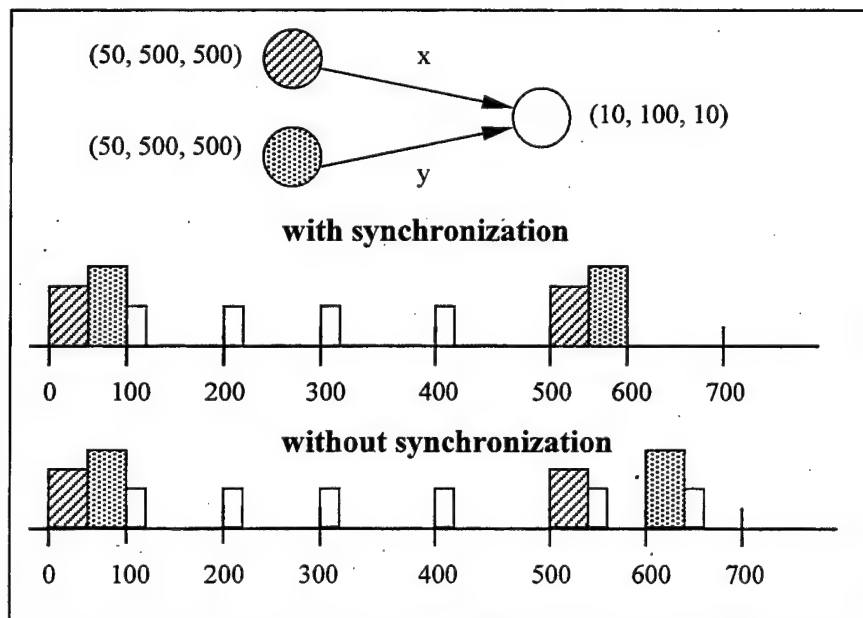
Figure 3.9. Reason for No Synch When $PER_{prod} \geq PER_{cons}$ for Uniprocessor Case

The only case in which $PER_{producer} < PER_{consumer}$ can be allowed is when there is no trigger at all. In this situation, synchronization is not needed, since it would place an additional burden on the scheduler and would not solve the problem of losing data. The

57

only advantage of having synchronization in this case is the fixed pattern for losing data. Furthermore, without explicit synchronization, the most that could happen (as seen in Figure 3.10), is that the consumer operator would read either the previous or the next instance of the data output by the producer. In other words, the operator would read the data one period apart at most.
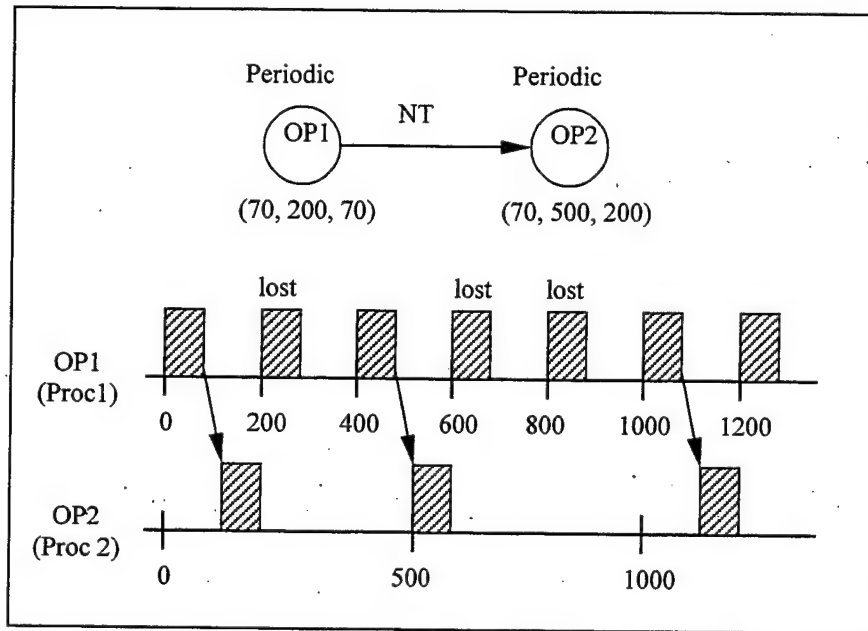
Figure 3.10. Reason for No Synch when PER$_{prod}$ < PER$_{cons}$ for Distributed Case

When PER$_{producer}$ ≥ PER$_{consumer}$, synchronization also does not solve the problem. It is possible to have two instances of the producer operator being scheduled one after the other, and this causes an overflow or loss of data depending on the triggering condition. This case is illustrated in Figure 3.11.

58
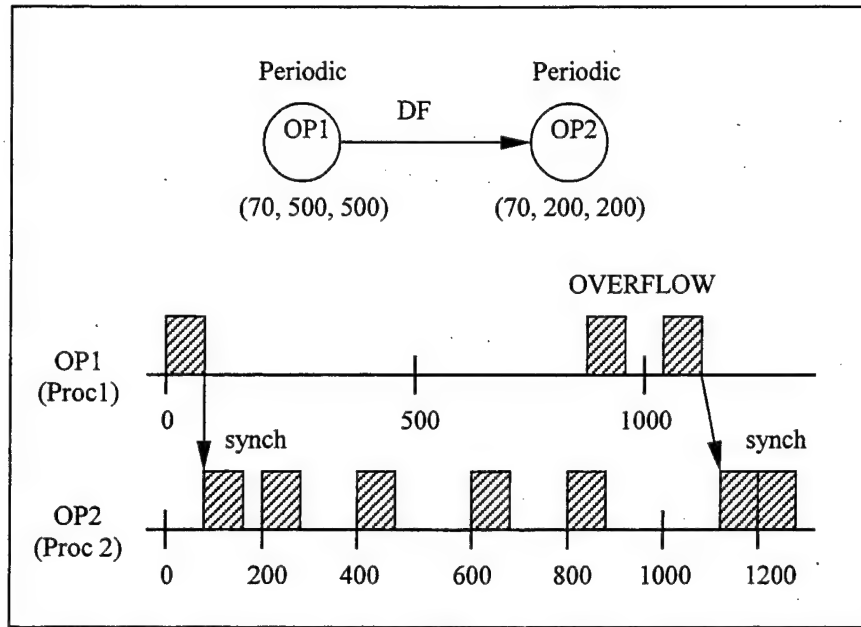
Figure 3.11. Reason for No Synch when $PER_{prod} \geq PER_{cons}$ for Distributed Case

As we can see, after scheduling each producer of a data flow or triggered by some sampled stream, the consumer of that data flow or sampled stream should be scheduled before the next instance of the producer. In a uniprocessor case, or even in a shared memory multiprocessor model, this approach is acceptable and easy to implement and guarantee. However, in a distributed case the lack of a master clock might cause a normally feasible schedule to become infeasible.

This assertion may be illustrated with a simple example. Assume a schedule for a two-processor system that meets all deadlines and synchronization requirements among their tasks, and that no buffer overflow occurs with respect to the data flow streams. Now, if clock drift occurs in processor 2, so that one of its consumers get shifted more than twice the period of its correspondent data flow producer, the consumer is guaranteed to lose data, and the schedule will fail (Cordeiro, 1995). Therefore, a new approach must be developed for the distributed case. Ideally, several sets of communicating processes would run independently in each processor, but with the guarantee that no data would be lost and no deadline missed (Cordeiro, 1995).

Since missing deadlines are always attached to data not being generated or consumed in the proper timing, we need to guarantee that all data being generated is consumed in a timely fashion. Then, the very first condition that must be satisfied is that $PER_{producer} \geq PER_{consumer}$, so that no data is lost.

59

2. Additional Restrictions Imposed on the Timing Constraints

In the previous subsection, we saw that for the distributed case synchronization is not needed. Obviously, there is a price to eliminate synchronization and that is a more stringent set of timing constraints for tasks.

Cordeiro (1995) shows that even with a faster consumer it is possible to have up to three occurrences of the slower producer between two consecutive instances of the consumer. Therefore, there can exist at most three instances of produced data waiting to be consumed at any instance of time. Also, based on this statement, we can say that any produced data will be consumed within at most two periods of the consumer, as demonstrated in Figure 3.12.
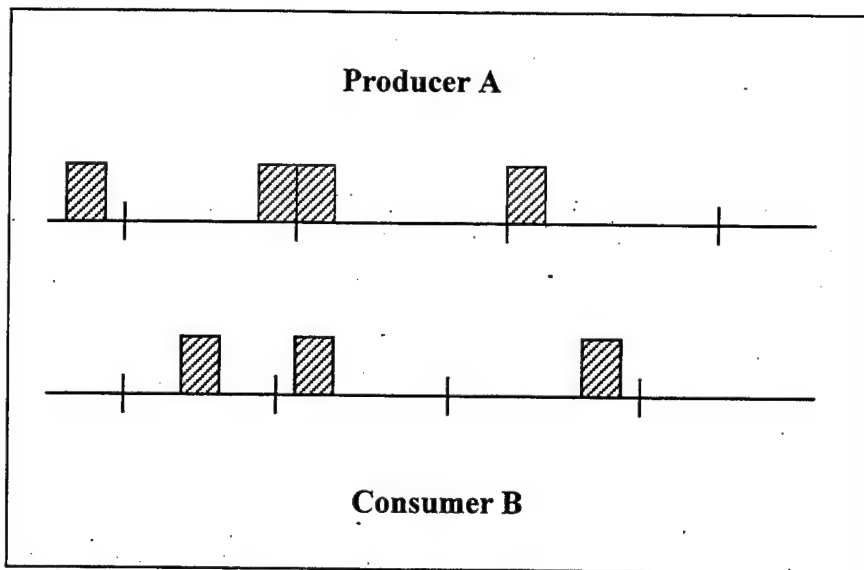
Figure 3.12. The Consumer-Producer Paradigm

It can be seen that the worst case that can occur is to have data from a producer consumed after $2 \times PER_{consumer} - MET_{consumer}$ units of time. Currently, in PSDL, contrary from the sporadic case, there is no upper bound on the time input data for a periodic operator should be consumed. Therefore, if the consumer is a periodic operator that receives data from network streams, not using synchronization will not impose any additional constraints on timing requirements.

In the sporadic case, however, the explicit upper-bound for consuming the incoming data is MRT, which is assumed to be greater than or equal to the latency plus the MET of the consumer operator for the incoming data. Therefore, an additional

restriction on the triggering period of a sporadic operator must be imposed when it has any data coming from network streams. Figure 3.13 shows the new timing constraints we should consider for the sporadic operator.
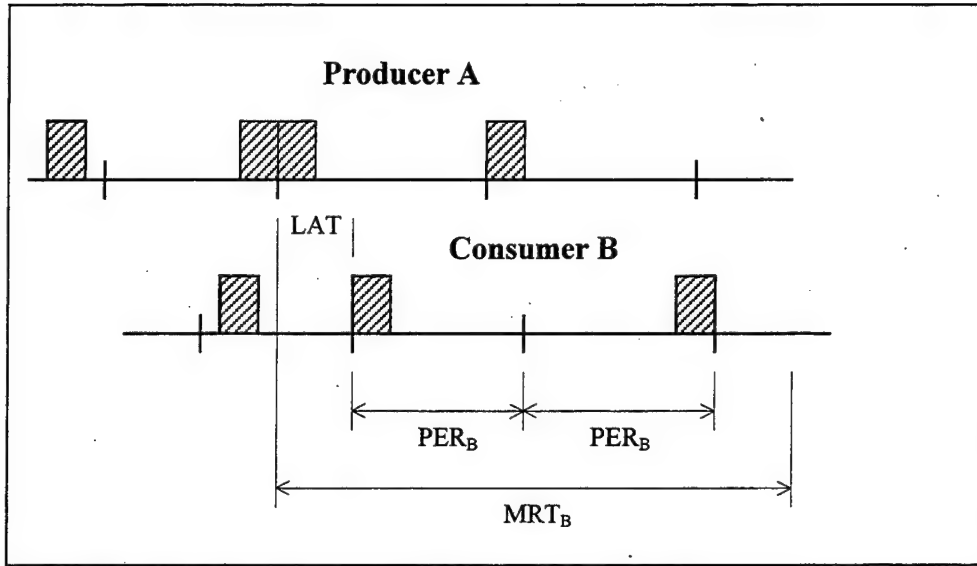
Figure 3.13. New Timing Constraints for the Sporadic Operator

From Figure 3.13, we have the following:

$$2 \times TP_B + LAT_{MAX} \leq MRT_B$$

or

$$TP_B \leq \frac{MRT_B}{2} - \frac{LAT_{MAX}}{2}$$

which is the new upper-bound for the triggering period of a sporadic operator. We also know, from the previous section, that $TP \geq MET$. Hence,

$$MET_B \leq TP_B \leq \frac{MRT_B}{2} - \frac{LAT_{MAX}}{2}$$

which is the new formula for calculating the triggering period of a periodic operator under the no synchronization assumption (Cordeiro, 1995).

IV. THE ALLOCATION AND SCHEDULING PROBLEM

A. INTRODUCTION

The task allocation and scheduling problem is one of the basic issues of building real-time applications on a distributed computing system (DCS). A DCS is typically modeled as a collection of processes interconnected by a communication network. For real-time applications, the allocation of tasks over the distributed system is intended to fully utilize the available processors, and the scheduling is needed to meet their timing constraints (Cheng and Agrawala, 1995).

A statically scheduled real-time system is composed of a number of real-time tasks dispatched according to the static schedule. Analysis is done *a priori* to determine the worst case condition, and the system is only deployed if the timing requirements are met. In this thesis, one of our objectives is to apply this approach to distributed real-time embedded systems. One of the most important problems with *a priori* analysis for real-time distributed systems has been complications introduced by communications costs: the delay for messages sent between processors must be accurately bounded.

In Chapter III, we saw a distributed scheduling analysis and how it is used to determine the conditions to guarantee that no deadlines are missed. In addition, we saw the additional restrictions imposed on the timing constraints due to the delay introduced by interprocessor communications under the no synchronization assumption. In this chapter, we focus on the scheduling problem in the Prototyping System Description Language (PSDL), which is a high-level prototyping language designed specifically to support conceptual modeling of real-time embedded systems.

The major difference between single processor and multiple processor scheduling is that, in addition to deciding which order to execute tasks, the multiple processor scheduling algorithms must decide which processor the task should run on. Therefore, we propose a technique to deal with the allocation problem.

As shall be shown, task allocation dramatically complicates the already complex problem of distributed software design, because in assigning m processes onto n processors, there are n^m different possible assignments. Optimal allocation is a problem of exponential complexity, and was proven to be NP-complete by Mok (Mok, 1983).

The key to process allocation is to establish an allocation model in terms of a cost function and additional constraints that match the application requirements as far as logical and timing correctness. The goal is to minimize the cost function under the constraints. Most of the cost functions found in available literature deal with performance. Others, such as those relating to reliability and fault tolerance, are only now emerging (Shatz and Wang, 1989).

The most widely used performance cost functions are:

1. Interprocessor communication cost (IPC) which is a function of the amount of data transferred, the network topology and link capacity;
2. Load balancing, which is a measure of how uniformly the workload is distributed among the processors. A good load balance will maximize system stability, which is the capability of busy hosts to receive bursty arrivals of processes without compromising their deadlines;
3. Completion time, the total execution time including interprocessor communication incurred by that processor.

The most frequent constraints found in typical real-time system are processor hardware limitations, dependence of some processes on certain processors, and number of available processors. The choice of a cost function obviously depends on the application, on the underlying hardware, and on several other characteristics (Cordeiro, 1995). Although distributed processing seems very attractive, one should be aware of the *saturation effect* that is sometimes forgotten by many developers. The basic consequence of this effect is that, contrary to expectations, the throughput does not increase linearly as the number of processors is increased. Actually, at some point (which can be as few as three or four processors) throughput starts to decrease (Chu, Holloway, Lan and Efe, 1980 and Jenny, 1977). The decrease in throughput is due to excessive interprocessor communication. Basically, the different approaches to solving the allocation problem all fall into one of three major classifications: graph theoretic, mathematical programming, or heuristics methods, which are by no means mutually exclusive.

The first of these represents the processes to be allocated as nodes in a graph, where each edge has a weight that is proportional to its inter-module communication cost (IMC), with the following remarks: an IMC of zero means that no communication takes place between those two modules and an IMC of infinity means that they should be assigned to the same processor. If a minimal-cut algorithm is performed on the graph, one ends up with the minimum allocation cost for those modules between two processors.

In general, however, an extension of this method to an arbitrary number of processors requires an n-dimensional min-cut flow algorithm, which quickly becomes computationally intractable (Cordeiro, 1995).

The mathematical programming approach uses, in most cases, the non-linear integer programming technique, where the above problem is formulated as a set of equations. It is very flexible in the sense that additional constraints can be included in the model very easily, moreover, this approach fails to accurately represent real-time constraints and precedence relations among tasks, because both factors introduce queuing delays into the system in a complex manner (Cordeiro, 1995).

Finally, heuristic methods (investigating practical ways of finding a solution), unlike the first two, try to find sub-optimal solutions for the assignment problem, which are generally faster, more extendible and simpler.

B. APPROACHING THE SCHEDULING PROBLEM

One of the major tasks when dealing with the scheduling problem is to determine whether the timing constraints of a given specification can be satisfied by some real-time systems. As we said above, analysis should be done *a priori* in order to determine the worst case condition. Therefore, we choose to demonstrate the schedulability of a prototype via generation of a static schedule that enforces all real-time constraints under the worst case conditions. Here we assume that each operator runs on the same processor from activation to completion. Thus, the CPU time is in one contiguous interval. We make this assumption to simplify the scheduling problem and, to keep interprocessor communication overhead at a minimum. All of the other timing constraints are bound on the duration of the time intervals defined by pairs of events.

Real-time systems typically consist of a mixture of periodic and sporadic tasks, each with an associated deadline and precedence constraints. Failure to meet critical task deadlines may lead to catastrophic failure of the system, requiring off-line analysis of allocation and processor scheduling to guarantee task deadlines (Tindell, Burns and Wellings, 1992). In this approach, the scheduler converts all sporadic time critical operators into equivalent periodic operators, as was shown in Chapter III. It is desirable to set the equivalent period as large as possible in order to minimize the impact on the load factor of the prototype. On the other hand, this increases the possibility of getting a

set of periods with a very large LCM (Least Common Multiple) after the conversion. According to Luqi and Shing (1996), prototypes with a large LCM are less likely to be schedulable. Hence a heuristic algorithm was developed to exploit the fact that a very small change in periods, while only affecting the load factor slightly, may be sufficient to dispose of some prime factors of the LCM, reducing it significantly. For further examination of this problem see Cordeiro (Cordeiro, 1995).

Based on what was explained above, we should assume that all the critical operators are periodic for the rest of the chapter. A set O of non-preemptive periodic operators with precedence relationship is schedulable if there exists a static schedule such that the start and completion time of every operator instance satisfies the timing and scheduling constraints. It is a well-known and accepted result that the least common multiple (LCM) of their periods provides a finite interval of time, for which a cyclic schedule can be calculated, if one exists, and repeated forever (Luqi and Shing, 1996). Based on the Harmonic Block Theorem that was shown in Chapter III, we can say that it suffices to compute the cyclic schedule within the interval $[0, 2 \times LCM]$. In Table 4.2 shows a summary of notations that will help the reader to get a better understanding of the scheduling problem.

Notation	Meaning
$act(o_{i,k})$	The beginning of the kth period of the operator o_i
$st(o_{i,k})$	The actual starting time of $o_{i,k}$
$ct(o_{i,k})$	The completion time of $o_{i,k}$
$d(o_{i,k})$	The deadline (latest completion time) of $o_{i,k}$
$tardiness(o_{i,k})$	The amount of time by which $o_{i,k}$ misses its deadline
$ready(o_{i,k})$	The earliest time when o_i can actually fires in the kth period

Table 4.1. Summary of Notations

C.   THE DISTRIBUTED SCHEDULING PROBLEM

Given a set of operators and a set of identical processors in a distributed system, a static schedule is a function that maps each instance of the operators that must start within $[0, 2 \times LCM]$ to a triple (P_{id}, st, ct), where P_{id} is the label of the processor that

66

executes the operator instance, st is the exact execution start time for the operator instance, and ct = st + MET, the time by which the operator instance must complete its execution. The reader should remember that here we assume that all of the instances of a given operator should be executed on the same processor.

A static schedule is said to be legal if the relative ordering of the operator-instances in the schedule satisfies the precedence constraints imposed by the prototype. A static schedule is said to be feasible if the schedule is legal and every operator-instance, when executed according to the schedule, meets its deadline. The cost of a schedule is defined here to be the maximum tardiness over all instances of the operators in G'. Hence, any legal schedule with zero cost is a feasible schedule.

The distributed static scheduling problem is to decide if there is a feasible schedule for the given scheduling constraint graph on a set of N processors. We already know that the static scheduling problem is NP-hard, so it is not likely to have efficient algorithms for solving the general static scheduling problem. Hence, we propose to construct an allocation model in terms of the interprocess communication cost, load balancing and timing correctness, so that the overall real-time constraints are matched.

Figure 4.1 shows the three kinds of graphs that are used by the CAPS scheduler to build the schedule. The global precedence graph G' can be obtained from the expanded dataflow graph G by removing all edges in G which represent state variables and then taking the transitive closure of the resultant graph. Given the global precedence graph, the scheduling constraint graph CG is defined by considering all task instances that must start execution in the interval [0, 2 × LCM], according to the algorithm for construction of the scheduling constraint graph presented by Luqi and Shing (1996).

Note that the scheduler does not construct the scheduling constraint graph explicitly. It computes the precedence constraints described by CG dynamically as it builds the static schedule based on the global precedence graph G' of the prototype (Luqi and Shing, 1996).

67

a) The Expanded Data flow Graph

b) The Global Precedence Graph

c) The Scheduling Constraint Graph
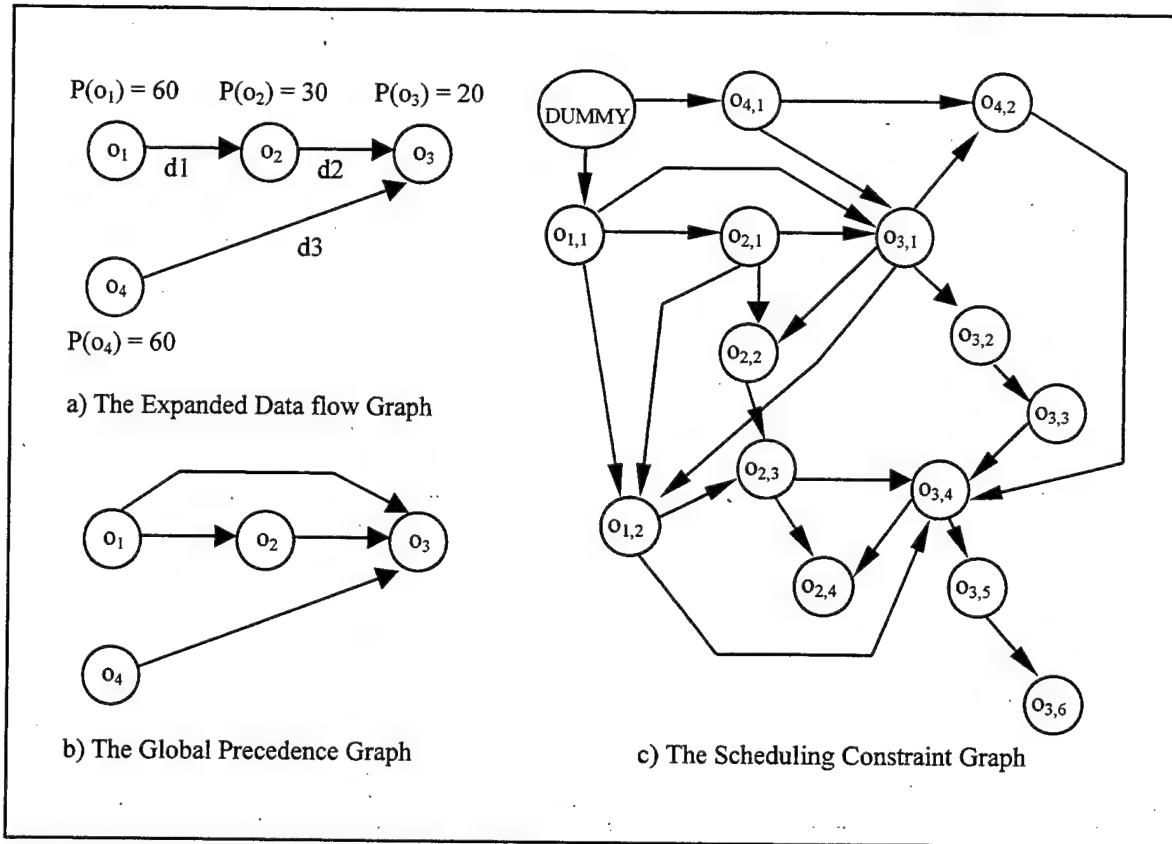
Figure 4.1. Graphs Used by the CAPS Scheduler

The scheduling constraint graph contains all task instances that must start within the interval $[0, 2 \times \text{LCM}]$. While defining CG, the scheduler is able to calculate the earliest time when operator-instance o_i can actually fire in the kth period, ready $(o_{i,k})$, and the deadline (latest completion time) of $o_{i,k}$, $d(o_{i,k})$. These notations are defined in (Luqi and Shing , 1996).

1.   Building the Distributed Schedule

Before starting to explain how the distributed schedule is built, we must mention the limitations of the algorithm we propose here. It only handles expanded dataflow graphs that contain cycles, if those cycles can be confined in each one of the available processors. Note that cycles in an expanded dataflow graph usually imply feedback loops of the state machines, where all operators in the loop are executed in locked steps and the input of each operator depends on the output of its previous firing. While the circular precedence constraints introduced by the cycles are broken by state streams in the cycle, the successive firing of the operators in the loop have to be synchronized to make sure that an operatoor acts upon the "feedbacks" (i.e. new input data) resulted from the

68

previous firing. While such synchronization is implied via the sequential execution of the operators in the static schedule for the uniprocessor case, explicit synchronization is needed for the distributed case where the operators in the feedback loop are assigned to different processors across the network, due to the long delays introduced by interprocessor communications. Another limitation is about operators connected by sampled steams running on different processors. We cannot always guarantee that a new data sent over the network will be consumed at the appropriate time by the consumer operator due to the lack of a global clock. For example, a drift in the clock of the processor on which the consumer operator is running may cause the consumer operator to lose the new data and, consequently, create dependency problems. To avoid these kinds of problems and make sure that underflow exceptions will not be raised we should use state streams to dissociate producer and consumer operators running on different processors.

The scheduling technique we propose here is based on ordered doubly linked list (DLL) structures. Although linked-list operations such as *traverse* or *insert in order* require O(N) time because they usually involve visiting each element in sequence, we can have considerable advantage in terms of flexibility. It is relatively easy to add new information by creating a new element and inserting it between two existing elements. Moreover, many of the operations in this model consist of grabbing information from the last element in the list, which requires O(1) time. However, the greatest advantage of using ordered doubly linked lists is that, when we finish the algorithm, the assignment of operators to processors and the static schedule are ready. There is no need to search the scheduling constraint graph. In addition, the tool that performs the automatic generation of code can easily access the information it needs, such as number of processors, interprocessor communications, allocation of operators to the available processors, etc.

We keep a doubly linked list for each processor and a separate list for the incoming communication network for each processor. Because operations on linked lists such as traverse and insert are expensive, initially each element in the list of a processor represents a time slot assigned to the first instance of an operator to be executed on that processor. Later, after a feasible partial schedule is found, these lists are extended, so that each element represents a time slot assigned to some instance of an operator. The

elements in each list of a processor P are ordered in increasing order of start time. Also, each element has the following fields:

1. Operator ID i
2. Operator-instance k
3. Activation time act
4. Start time st
5. Completion time ct
6. Deadline d
7. Number of instances that start within $[0, 2 \times LCM]$ n_i

For the communication network list, each element represents an incoming communication arriving at the corresponding processor and the list is ordered in increasing order of latency. Each element of a communication network list has the following fields:

1. Producer o_i
2. Consumer o_j
3. Latency $L(o_i \rightarrow o_j)$

We know that distributing tasks to as many processors as possible tends to increase the communication delay, which decreases the chances for the scheduler to find a feasible schedule. So, there is a trade-off between the advantages of maximizing parallelism (maximize the number of operators executing in parallel) and minimizing communication delay. Although maximum parallelism seems to be good at the beginning, it increases interprocessor communication demand and, consequently, the overall communication delay. Therefore, our objective here is to keep a reasonable number of processors, but still take advantage of concurrent execution.

A set of periodic operators in a prototype is not schedulable if the load factor of the protoype is greater than the number of available processors. In other words, the number of available processors should be, at least equal to $\lceil \Sigma \; MET(x)/PER(x) \rceil$ over all periodic operators x. For example, if the load factor is equal to 1.8, then for the protoype to be schedulable we need at least $\lceil 1.8 \rceil = 2$ processors. However, a high load factor may affect the schedulability of the prototype. Therefore, to increase the efficiency of the algorithm we set the value *max_load_factor,* which is the maximum allowed load factor for each processor.

Assume *max_load_factor* = 0.7 in the example above. Now, for the prototype to be schedulable we need $\lceil 2/0.7 \rceil = 3$ processors. Therefore, considering the explanation

70

above, we decide to keep the number n of processors equal to max(z, w) where z is the minimum number of processors required by the prototype when considering the load factor and *max_load_factor*, and w is the number of operators with no incoming edge.

Given the global precedence graph G' of the prototype, the partial distributed static schedule is built as following:

1. Calculate n = max(z, w). Then create n linked lists, which represent the n processors, and n linked lists representing the communication network.
2. For each operator with no incoming edge o_i in G', $1 \le i \le w$, create a copy of the operator and insert it into the list corresponding to processor P_i, where i = (1, 2, ..., w). The operator is assigned st = 0 when inserted into the first position in the list for processor P_i.
3. For each list corresponding to a processor P_i, $1 \le i \le w$, remove the children of the first operator o_i in G' and insert them into the Ready_Set. Next, while Ready_Set is not empty, remove the operator with the earliest start time and calculate the load factor for the processor P_i. If load factor < *max_load_factor*, then insert the operator in the list. If load factor ≥ *max_load_factor*, then start to fill the list for the next available processor.

At this point, we have allocated each operator in the prototype to a processor and built the partial schedule. Now we should adapt the partial schedule to the distributed model, that is, modify the start time and completion time of all the operators to include the delay introduced by each interprocessor communication. For any two vertices o_i and o_j in G', if edge $o_i \to o_j$ exists in G and o_i is located on a different processor than o_j, then pick up the edge $o_i \to o_j$ with the greatest latency $L(o_i \to o_j)$ in G, calculate the latency L for that specific interprocessor communication and, in the case $L > L(o_i \to o_j)$, assign the value of L to $L(o_i \to o_j)$. Now, if $ct(o_i) + max[L(o_i \to o_j)] > st(o_j)$, then we must update the start time and completion time of operator o_j so that.

$$st(o_j) = ct(o_i) + max[L(o_i \to o_j)]$$

and

$$ct(o_j) = st(o_j) + MET_j$$

In addition, the start time of all operators on the same processor with a lower precedence constraint than o_j that are affected by the communication delay should be updated accordingly, so that the precedence and mutual exclusion constraints are satisfied as originally required. Finally, create a new element to be inserted into the

communication network list corresponding to the processor where o_j is located. Figure 4.2 shows the algorithm used to adapt the schedule to the distributed model.

```
Procedure Adapt_Schedule_To_Distributed_Model
                              (Partial_Schedule, Communication_Network) is
Begin
Max_Latency := 0;
For i in 1 to Number of Vertex in G' loop
    For j in 1 to Number of Vertex in G' loop
        If i /= j then
            If Processor_Id(i) /= Processor_Id(j) then
                For each edge o_i → o_j in G' loop
                    If edge o_i → o_j is present in G then
                        If Latency(o_i → o_j) > Max_latency then
                            Max_latency := Latency(o_i → o_j);
                        End if;
                    End if;
                End loop;
                If (IPC_Latency := Calculate_IPC_Latency(o_i → o_j)) > Max_Latency then
                    Max_Latency := IPC_Latency;
                End if;
                Producer_Ptr := Find_Operator(i, Partial_Schedule(Processor_ID(i)));
                Consumer_Ptr := Find_Operator(j, Partial_Schedule(Processor_ID(j)));
                If (Producer_Ptr.ct + Max_Latency) > Consumer_Ptr.st then
                    Consumer_Ptr.st := Producer_Ptr.ct + Max_Latency;
                    Consumer_Ptr.ct := Consumer_Ptr.st + MET(j);
                    Lower_Precedence_Op := Consumer_Ptr.Next;
                    While Lower_Precedence_Op /= Null loop
                        If Lower_Precedence_Op.st
                                            < Lower_Precedence_Op.Previous.ct then
                            Lower_Precedence_Op.st :=
                                            Lower_Precedence_Op.Previous.ct;
                            Lower_Precedence_Op.ct := Lower_Precedence_Op.st +
                                            MET(Lower_Precedence_Op.i);
                        End if;
                        Lower_Precedence_Op := Lower_Precedence_Op.Next;
                    End loop;
                End if;
                Net_Communication := New_Net_Element(i, j, Max_latency);
                Insert_in_Order
                        (Net_Communication, Communication_Network(ProcessorID(j)));
            End if;
        End if;
    End loop;
End loop;
End;
```

Figure 4.2. Adapt Schedule to the Distributed Model Algorithm

The reader should remember that, according to the analysis for the distributed scheduling we saw in Chapter III, to guarantee that no data is lost, the period of operator o_i must be greater than or equal to the period of operator o_j. So, a warning must be displayed by the scheduler in case this condition is not satisfied.

It should have been noticed that the scheduler uses the earliest-starting-time-first algorithm to build the schedule. It removes the vertex with the earliest start time among all the vertices v in the Ready_Set and inserts it to a list. According to Luqi and Shing (1996), the earliest-starting-time-first algorithm is very effective in finding a feasible solution for prototypes with load factor 0.7 or below, so 0.7 is a good reference for the value of *max_load_factor* for each processor.

The major difference between distributed scheduling and multiprocessor scheduling is the delay introduced by interprocessor communications. The latency for the transmission of a data value b bits long from processor P_i to P_j is considered to be zero in a shared memory, multiple processor configuration. On the other hand, the latency is $D_{i,j} + (b \times T_{i,j})$ for the distributed configuration, where:

1. $D_{i,j}$ is the interprocessor delay, and
2. $T_{i,j}$ is the inverse of the link speed (seconds per bit).

Another difference is that the scheduler initially allocates only the first instance of the operators to processors. Later, after a feasible partial schedule is found, the partial schedule is extended (see Figure 4.3), so that the elements in the list for each processor represent the operator-instances that start execution within $[0, 2 \times LCM]$. Because of this fact, we need to come up with a way to extend the schedule, which is done as following:

1. When a new element is inserted into the list for each processor, we calculate the number n_i of instances of the operator o_i that start within the interval $[0, 2 \times LCM]$, where $n_i = \dfrac{2 \times LCM}{PER_i}$;

2. After a feasible partial schedule is found, we should traverse each list and for each element insert $n_i - 1$ new elements in the list, so that all of the operator-instances which start execution within $[0, 2 \times LCM]$ have a corresponding element in the list.

```
Function Extend_Partial_Schedule(Partial_Shedule) is
Begin
    For j in 1 to Number_of_Processors(Partial_Schedule) loop
        Current_Operator_Ptr := Partial_Schedule(j).Head;
        Last_Operator_Ptr := Partial_Schedule(j).Tail;
        While Current_Operator_Ptr /= Last_Operator_Ptr.Next loop
            For count in 1 to Current_Operator_Ptr.n_i − 1 loop
                New_Operator_Instance_Ptr := Create_New_Element;
                New_Operator_Instance_Ptr.i := Current_Operator_Ptr.i;
                New_Operator_Instance_Ptr.k := 1 + Current_Operator_Ptr.k;
                New_Operator_Instance_Ptr.act := Current_Operator_Ptr.st +
                                    count × PER(New_Operator_Instance_Ptr.i);
                New_Operator_Instance_Ptr.d := New_Operator_Instance_Ptr.act +
                                    FW(New_Operator_Instance_Ptr.i);
                Insert_in_Order(New_Operator_Instance_Ptr, Partial_Schedule(j));
                -- Insert new element to the list in increasing order of start time.
                -- The actual st and ct of the new operator instance is calculated after it is
                -- inserted in the list, according to the ct of the previous operator, such
                -- that the mutual exclusion constraint is satisfied. Also, we must ensure
                -- that the start time of the next operator is greater than or equal to ct of
                -- the new element.
            End loop;
            Current_Operator_Ptr := Current_Operator_Ptr.Next;
        End loop;
    End loop;
    Return(Partial_Schedule);
End;
```

Figure 4.3. Extend Partial Schedule Algorithm

2. Verifying the Feasibility of the Schedule

As we saw above, the scheduler first builds a partial schedule by allocating just the first instance of each operator to the available processors. Then, it adapts the schedule to the distributed model, which means that all of the operators affected by communication delay due to interprocessor communication (more specifically, the consumer operator and probably other operators with lower precedence constraints) must have their start times updated. By updating the start time of those operators affected by communication delay, the scheduler guarantees that the consumer operator is able to read the data sent by the incoming interprocessor communication and the precedence constraints are still satisfied.

75

Next, we must verify if the partial schedule is feasible, that is, we must verify that every operator executed according to the schedule will meet its deadline. We do this by evaluating the *cost* of the partial schedule. If the partial schedule satisfies the precedence constraints imposed by G' and its cost is zero, then the partial schedule is feasible. Figure 4.4 shows the algorithm used to evaluate the *cost* of the distributed schedule. After evaluating the cost of the partial schedule, we check the value returned by function Evaluate_Cost, and if cost is equal to zero, then a feasible partial schedule was found. Now, the partial schedule is ready to be extended.

```
Function Evaluate_Cost(Schedule; Number_of_Processors)
Begin
    Expected_Lower_Bound := 0;
    For j in 1 to Number_of_Processors loop
        Lower_Bound(j) := max{0, ct(o_{i,k}) - d(o_{i,k}) | for all operator-instances o_{i,k} in the
                        list for processor P_j in the Schedule}
        If  Lower_Bound(j) > 0 then
            return(i);
        End if;
    End loop;
    return(Expected_Lower_Bound);
End;
```

Figure 4.4. Evaluating the Cost of a Distributed Schedule

As we can see in Figure 4.4, the algorithm checks each processor to determine if there is any operator that does not meet its deadline. If there is an operator missing its deadline, the function Evaluate_Cost returns the identification of that processor. Next, the distributed scheduling algorithm outputs message "Schedule Not found" and exits.

After a feasible partial schedule is found, the schedule is extended as explained in the previous subsection. The activation time of each new element inserted in the list is calculated as following:

$$act(o_{i,k}) = st(o_{i,1}) + (k - 1) \times PER(o_i)$$

According to the PSDL mutual exclusion constraint (see Section A.5 in Chapter III), the execution time of operator-instances running on the same processor cannot overlap. Therefore, after inserting a new operator in the list ordered by start time, we must check if its activation time is greater than or equal to the completion time of the overlapping operator. If it is not, the start time of the new operator will be assigned the value ct of the

previous operator. Also, we must ensure that completion time of the new operator is less than or equal to the start time of the next operator in the list. If it is not, then we must update the start time and completion time of all the affected operators in the list. Finally, we should verify the feasibility of the extended schedule by evaluating the cost of the schedule. If the value returned by function Evaluate_Cost is equal to zero, a feasible schedule was found. Figure 4.5 below shows the distributed scheduling algorithm.

```
The Distributed_Scheduling Algorithm
Begin
      Max_Load_factor := 0.7;
      z := Minimal_Number_Of_Processors;
      w := Number_Of_Operators_With_No_Incoming_Edge;
      Number_Of_Processors := max(z, w);
      Partial_Schedule(1..Number_Of_Processors);
      Communication_Network(1..Number_Of_Processors);
      Ready_Set = { };
      For each operator with no incoming edge in G' loop
            v := Operator_With_No_Incoming_Edge;
            Add_Item_To_Set(v, Ready_Set);
      End loop;
      count := 1;
      While Not_Empty(Ready_Set) loop
            v := Remove_Item_With_Earliest_Start_Time(Ready_Set);
            operator_instance(v) := 1;
            st(v) := 0;
            ct(v) := st(v) + MET(v);
            d(v) : deadline(v);
            Add_Item_To_Schedule(v, Partial_Schedule(count));
            count := count + 1;
      End loop;
      For i in 1 to Number_Of_Processors loop
            v := First_Element(Partial_Schedule(i));
            For each child u of v remaining in G' loop
                  Remove_Child_And_Add_To_Set(u, Ready_Set);
            End loop;
            count := i;
            While Not_Empty(Ready_Set) loop
                  If  Load_Factor(Partial_Schedule(count)) < Max_Load_Factor then
                        v := Remove_Item_With_Earliest_Start_Time(Ready_Set);
                        operator_instance(v) := 1;
                        st(v) := max{Last_Stop_time(Partial_Schedule(count)), ready(v)};
                        ct(v) := st(v) + MET(v);
                        d(v) := deadline(v);
                        Add_Item_To_Schedule(v, Partial_Schedule(count));
                  Else
                        count = count + 1;
                  End if;
            End loop;
      End loop;
      Adapt_Schedule_To_Distributed_Model
                              (Partial_Schedule, Communication_Network);
```

```
        If (Evaluate_Cost(Partial_Schedule, Number_Of_Processors) > 0 then
                Output "Schedule Not Found";
        Else
                Extended_Schedule := Extend_Partial_Schedule(Partial_Schedule);
                If (Evaluate_Cost(Extended_Schedule, Number_Of_Processors) = 0) then
                        Output "Schedule Found";
                Else
                        Output "Schedule Not Found";
                End if;
        End if;
End;
```

Figure 4.5. The Distributed Scheduling Algorithm

The following example illustrates how the algorithm shown in Figure 4.5 works. Consider the expanded dataflow graph in Figure 4.1 (a) and assume that the operators in G have the following timing constraints: $o_1(50, 600, 600)$, $o_2(70, 300, 300)$, $o_3(50, 200, 200)$ and $o_4(50, 600, 600)$. Also, assume that all the latencies are equal to zero. Given the expanded dataflow graph G, the PSDL scheduler builds the global precedence graph G' and the scheduling constraint graph CG as shown in Figure 4.1 (b) and (c), respectively, and defines the ready time and deadline for each operator-instance that starts execution within the interval [0, 1200].

Following the distributed scheduling algorithm above, the number of processors is calculated according to the load factor of the prototype and the number of operators without an incoming edge. In this case, the values are 0.65 and 2, respectively. Thus, considering *max_load_factor* equal to 0.7, the minimal number of processors is two. Therefore, the number of processors for the distributed model is two as well. Next, the partial schedule is built, and looks like Figure 4.6.

79
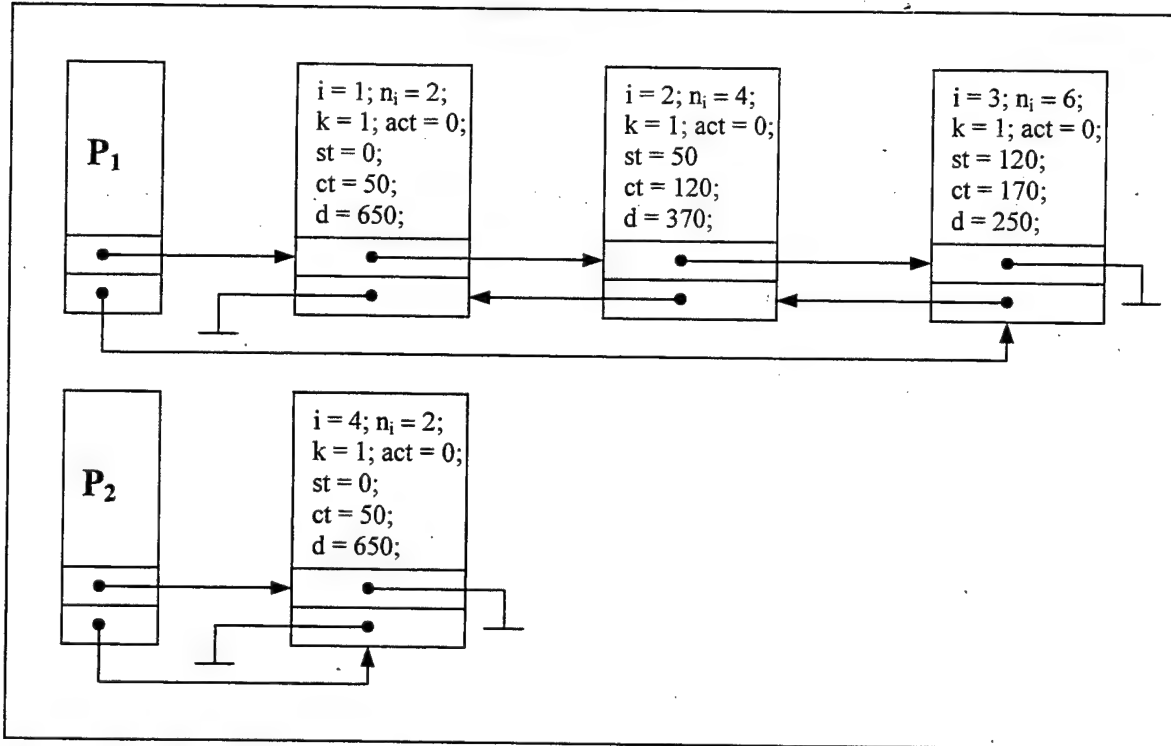
Figure 4.6. Partial Schedule

The partial schedule is now adapted to the distributed model and the feasibility of the schedule is verified. Note that during the adapting phase the scheduler creates a new element in the communication network list for processor P_1, which corresponds to the interprocessor communication between operator o_4 in P_2, and operator o_3 in P_1. Assuming that the latency L for the interprocessor communication from o_4 to o_3 is equal to 50 ms, then the value of $L(o_4 \rightarrow o_3)$ becomes 50. Since $ct(o_4) + L(o_4 \rightarrow o_3)$ is less than $st(o_3)$, the start time of operator o_3 is not changed when adapting the partial schedule. As we can see in Figure 4.6, the cost of the partial schedule is zero because no operator misses its deadline, even under the distributed model.

Next, the partial schedule is extended, so that all the operator-instances in the prototype which start execution within the interval [0, 1200] have a corresponding element in the schedule. As we can see in Figures 4.7 (a) and (b), the scheduler found a feasible distributed schedule.

Operator-Instance	Start Time	Completion Time	Deadline	Tardiness
$O_{1,1}$	0	50	650	0
$O_{2,1}$	50	120	370	0
$O_{3,1}$	120	170	250	0
$O_{2,2}$	350	420	650	0
$O_{3,2}$	420	470	520	0
$O_{3,3}$	550	600	720	0
$O_{1,2}$	600	650	1200	0
$O_{2,3}$	650	720	950	0
$O_{3,4}$	750	800	920	0
$O_{2,4}$	950	1020	1250	0
$O_{3,5}$	1020	1070	1120	0
$O_{3,6}$	1150	1200	1320	0

Figure 4.7 (a). Static Schedule for Processor P_1

Operator-Instance	Start Time	Completion Time	Deadline	Tardiness
$O_{4,1}$	0	50	650	0
$O_{4,2}$	600	650	1200	0

Figure 4.7 (b). Static Schedule for Processor P_2

81

V.   SOFTWARE ARCHITECTURE DESIGN

A.   INTRODUCTION

The Ada 95 programming language contains a distributed systems annex that defines facilities for supporting the implementation of distributed systems using multiple partitions working cooperatively as part of a single Ada program (Ada, 1995). The Distributed Systems annex of the Ada 95 Language Reference Manual (Ada, 1995) defines a distributed system as an interconnection of one or more processing nodes (a system resource that has both computational and storage capabilities), and zero or more storage nodes (a system resource that has only storage capabilities, with the storage addressable by one or more processing nodes). The process of mapping the partition of a program to the nodes in a distributed application is called configuring the partitions of the program.

In this chapter, we will discuss the characteristics and capabilities of Ada 95 distributed systems and present the software GLADE (GNAT Library for Ada Distributed Execution). GLADE is an implementation of the Distributed Systems annex for the GNAT compiler. GLADE has been developed by Ada Core Technologies and a research team from Ecolé National Superior de Telecommunication (Paris and Brittany, France). A complete reference can be found in GLADE User Manual (ACT Europe).

1.   Partitions

According to the Ada 95 Language Reference Manual (Ada, 1995), the partitions of a distributed program are classified as either active or passive. An active partition is a program or part of a program that can be invoked from outside the Ada implementation. For example, a partition might be an executable file generated by the system linker. The user can explicitly assign library units to a partition. The compilation units included in a partition are those of the explicitly assigned library units, as well as other compilation units needed by those library units. An active partition shall be configured on a processing node. The user can optionally designate one subprogram as the main subprogram for the partition. A main subprogram, if specified, shall be a subprogram.

A passive partition is a partition that has no control threads of its own, whose library units are all pre-elaborated, and whose data and subprograms are accessible to one

or more active partitions. A library unit is pre-elaborated if a pragma Pre-elaborate applies to the library unit. If a library unit is pre-elaborated, then its declaration and body (if any) are elaborated prior to all elaborated library items of the partition. A passive partition shall be configured either on a storage node or on a processing node. The configuration of the partitions of a program onto a distributed system shall be consistent with the possibility for data references or calls between the partitions implied by their semantic dependencies. Any reference to data or call of a subprogram across partitions is called a remote access (Ada, 1995).

2.   Categorization of Library Units

Library units can be categorized according to the role they play in a distributed program. Certain restrictions are associated with each category to ensure that the semantics of a distributed program remain close to the semantics for a nondistributed program. A categorization pragma is a library unit pragma that restricts the declarations, child units, or semantic dependencies of the library unit to which it applies. A categorized library unit is a library unit to which a categorization pragma applies. The following pragmas are examples of categorization pragmas: Shared_Passive, Remote_Types, and Remote_Call_Interface. A normal library unit is one to which no categorization pragma applies.

According to the Distributed Systems annex of the Ada 95 Language Reference Manual (Ada, 1995), the following defines each categorized library unit by categorization pragma:

1. Shared Passive Library Unit
   A shared library unit is used for managing global data shared between active partitions. The restrictions on shared passive library units prevent the data or tasks of one active partition from being accessible to another active partition through references implicit in objects declared in the shared passive library unit.
2. Remote Types Library Unit
   A remote types library unit supports the definition of types intended for use in communication between active partitions.
3. Remote Call Interface Library Unit
   A remote call interface library unit can be used as an interface for remote procedure calls (RPCs) between active partitions.

84

3.   Remote Subprograms Calls

A remote subprogram call is a subprogram call that invokes the execution of a subprogram in another partition. The partition that originates the remote subprogram call is the calling partition, and the partition that executes the corresponding subprogram call is the called partition. Some remote procedure calls are allowed to return prior to the completion of program execution. These are called asynchronous remote procedure calls. (Ada, 1995)

There are three different ways of performing a remote subprogram call:

1. As a direct call on a (remote) subprogram explicitly declared in a remote call interface;
2. As an indirect call through a value of a remote access-to-subprogram type;
3. As a dispatching call with a controlling operand designated by a value of a remote access-to-class-wide type.

The first way of calling corresponds to a static binding between the calling and the called partition. The latter two ways correspond to a dynamic binding between the calling and the called partition.

A remote call interface library unit defines the remote subprograms or remote access types used for remote subprogram calls. For the execution of a remote subprogram call, subprogram parameters (and later the results, if any) are passed using a stream-oriented representation which is suitable for transmission between partitions. This action is called marshalling. Unmarshalling is the reverse action of reconstructing the parameters or results from the stream-oriented representation. Marshalling is performed initially as part of the remote subprogram call in the calling partition; unmarshalling is done in the called partition, results (if any) are marshalled in the called partition, and finally unmarshalling is done in the calling partition. (Ada, 1995)

A calling stub is the sequence of code that replaces the subprogram body of a remotely called subprogram in the calling partition. A receiving stub is the sequence of code that receives a remote subprogram call on the called partition and invokes the appropriate subprogram body. The task executing a remote subprogram call blocks until the subprogram in the called partition returns, unless the call is asynchronous. For an asynchronous remote procedure call, the calling task can become ready before the procedure in the called partition returns.

85

If a construct containing a remote procedure call is aborted, the remote subprogram call is cancelled. If an exception is propagated by a remotely called subprogram, and the call is not an asynchronous call, the corresponding exception is raised again at the point of the remote subprogram call. For an asynchronous call, if the remote procedure call returns prior to the completion of the remotely called subprogram, any exception is lost. The exception Communication_Error is raised if a remote call cannot be completed due to difficulties in communicating with the called partition.

4. Partition Communication Subsystem

The Partition Communication Subsystem (PCS) provides facilities for supporting communication between the active partitions of a distributed program. The package System.RPC is a language-defined interface to the PCS. An implementation conforming to Annex E of the Ada Language Reference Manual shall use the RPC interface to implement remote subprogram calls (Ada, 1995). Figure 5.1 shows the specification for the package System.RPC.

```
with Ada.streams;
package System.RPC is
    type Partition_ID is range 0..implementation-defined;
    Communication_Error : exception
    type Params_Stream_type (
        Initial_Size : Ada.Streams.Stream_Element_Count) is new
        Ada.Streams.Root_Stream_Type with private;


    procedure Read (Stream : in out Params_Stream_Type;
                    Item : out Ada.Streams.Stream_Element_Array;
                    Last : out Ada.Streams.Stream_Element_Offset);

    procedure Write (Stream : in out Params_Stream_Type;
                    Item      : in Ada.Streams.Stream_Element_Array);

    -- Synchronous call
    procedure Do_RPC (Partition : in Partition_ID;
                    Params    : access Params_Stream_Ttype;
                    Result    : access Params_Stream_Ttype);

    -- Asynchronous call
    procedure Do_APC (Partition : in Partition_ID;
                    Params    : access Params_Stream_Ttype);

    --The handler for incoming RPCs
    type RPC_Receiver is access procedure (Params : access Params_Stream_type;
                                        Result  : access Params_Stream_type);

    procedure Establish_RPC_Receiver (Partition : in Partition_ID;
                                    Receiver : in RPC_Receiver);

private
    -- not specified by the language
end System.RPC;
```

Figure 5.1 Specification of Package System.RPC (Ada, 1995)

As we can see in Figure 5.1, a value of the type Partition_ID is used to identify a partition. During the execution of a remote procedure call, subprogram parameters (and later results, if any) are passed using a stream-oriented representation which is suitable for transmission between partitions. As we saw above, the annex calls this action marshalling. Unmarshalling is the reverse action of reconstructing the parameters or results from the stream-oriented representation.

87

According to the Distributed Systems annex of the Ada 95 Language Reference Manual (Ada, 1995), an object of the type Params_Stream_Type is used for identifying the particular remote subprogram that is being called, as well as marshalling and unmarshalling the parameters or result of a remote subprogram call as part of sending them between partitions. The Read and Write procedures override the corresponding abstract operations for the type Params_Stream_Type.

Both synchronous and asynchronous communications are supported by package System.RPC and are implemented by the procedures Do_RPC and Do_APC, respectively. Both procedures send a message to the active partition identified by the Partition parameter. After sending a message, Do_RPC blocks the calling task until a reply message comes back from the called partition or some error is detected by the underlying communication system. In this case, Communication_Error is raised at the point of the call to Do_RPC. Do_APC operates in the same way as Do_RPC, except that it is allowed to return immediately after sending a message.

Finally, if the partition includes a Remote_Call_Interface (RCI) library unit, the procedure Establish_RPC_Receiver is called once, immediately after elaborating the library units of an active partition but prior to invoking the main subprogram. The receiver parameter designates an implementation-provided procedure called the RPC_Receiver, which will handle all RPCs received by the partition. Establish_RPC_Receiver saves a reference to the RPC_Receiver. When a message is received at the called partition, the RPC_Receiver is called with the Params stream containing the message. When the RPC_Receiver returns, the contents of the stream designated by result is placed in a message and sent back to the calling partition. The implementation of the RPC_Receiver shall be reentrant, thereby allowing concurrent calls on it from the PCS to service concurrent remote subprogram calls into the partition. (Ada, 1995)

5.   The Package Streams

A Stream is a sequence of elements comprising values from possibly different types and allowing sequential access to these values. A stream type is a type in the class whose root type is Streams.Root_Stream_Type. (Ada, 1995)

88

The types in this class represent different kinds of streams. The pre-defined stream-oriented attributes like T'Read and T'Write makes dispatching calls on the Read and Write procedures of the Root_Stream_Type. Figure 5.2 shows the specification of package Ada.Streams.

```
Package Ada.Streams is
    pragma Pure (stream);
    type Root_Stream_Type is abstract tagged limited private;
    type Stream-Element is mod implementation-defined;
    type Stream_Element_Offset is range implementation defined;
    subtype Stream_Element_Count is
            Stream_Element_Offset range 0..Stream_Element_Offset'Last;
    type Stream_Element_Array is
            array (Stream_Element_Offset range <>) of Stream_Element;

    procedure Read (Stream : in out Root_Stream_Type;
                    Item    : out Stream_Element_Array;
                    Last    : out Stream_Element_Offset) is abstract;

    procedure Write (Stream : in out Root_Stream_Type;
                    Item    : in Stream_Element_Array) is abstract;
private
    --not defined by the language
end Ada.Streams;
```

Figure 5.2. Specification of Package Ada.Streams (Ada, 1995)

The read operation transfers Item'Length stream elements from the specified stream to fill the array Item. The index of the last stream element transferred is returned in Last. Last is less than Item'Last only if the end of the stream is reached. The Write operation appends Item to the specified stream. There are also the stream-oriented attributes Read, Write, Output, and Input that convert values to a stream of elements and reconstruct values from a stream. For every subtype S of a specific type T, some attributes are defined, which denotes a procedure or a function call. Figure 5.3 presents those attributes.

89

```
procedure S'Write (Stream : access Ada.Streams.Root_Stream_Type'Class;
                   Item   : in T);
-- S'Write writes the value of Item to Stream

procedure S'Read (Stream : access Ada.Streams.Root_Stream_Type'Class;
                  Item   : out T);
-- S'Read reads the value of Item from Stream

procedure S'Output (Stream : access Ada.Streams.Root_Stream_Type'Class;
                    Item   : in T);
-- S'Output writes the value of Item to Steram, including any bounds or
-- discriminants

function S'Input (Stream : access Ada.Streams.Root_Stream_Type'Class)
                 return T;
-- S'Input reads and returns the value of Item from stream, using any bounds
-- or discriminants written by a corresponding S'Output
```

Figure 5.3. Stream Attributes (Ada, 1995)

B.   ANALYSIS OF GLADE AND ITS CONFIGURATION LANGUAGE

Ada 95 is the first general-purpose language to provide a standard distributed programming paradigm. By combining the distributed and object-oriented features of Ada 95, it is possible to create an application where objects are physically distributed over a network of machines without having to interface to any low-level communication layer. Likewise, by combining the distributed and real-time capabilities of Ada 95, it is possible to design applications which meet real-time constraints in a distributed environment (ACT Europe).

GLADE release 1.03p for GNAT 3.10p distribution contains different components organized in separate directories:

1. Garlic: a PCS (Partition Communication Subsystem) which is a high level communication layer that provides several classical services available in a distributed system (partition identification management, name services, distributed termination, etc.) and that accommodates several network protocols and communication subsystems;
2. Dist: a partitioning tool called "gnatdist" and its configuration language which allow you to divide your program into a number of independent partitions and specify the machine where the individual partitions are to execute;
3. Ada: a subset of GNAT sources necessary to build GLADE.

GLADE is a complete environment for developing distributed applications that includes:

1. a complete PCS (Partition Communication System);
2. a simple partition description language and tool;
3. utilities to build and start distributed applications.

The caller and receiver stubs needed by the PCS are generated by the GNAT compiler using special flags. The partitioning tool is responsible for calling GNAT with the appropriate flags.

An Ada 95 distributed application comprises a number of partitions which can be executed concurrently on the same machine or, can be distributed on a network of machines. The way in which partitions communicate was described in the previous section. A partition is a set of compilation units which are linked together to produce an executable binary. A distributed program comprises two or more communicating partitions. The distributed systems annex does not describe how a distributed application should be configured. It is left to the user to define the partitions in his program and on which machine they should be executed.

1. Configuring a Distributed Application

The tool gnatdist and its configuration language have been purposely designed to allow partitioning a program and to specify the machines where the individual partitions are to execute on. Gnatdist reads a configuration file and builds several executables, one for each partition. It also launches the different partitions and passes arguments specific to each partition.

The GLADE User Manual (ACT Europe) describes the following steps that should be followed to configure a distributed application:

1. Write a non-distributed Ada application. Use the categorization pragmas Remote_Call_Interface and Remote_Types to specify the packages that can be called remotely (the Shared_Passive categorization pragma is not yet implemented).
2. When this non-distributed application is working, write a configuration file that maps your categorized packages onto partitions. Do not forget to specify the main procedure of your distributed application as explained below.
3. Type 'gnatdist configuration-file'. The gnatdist command line options are:
   gnatdist [switches] configuration-file [list of partitions]
   The switches of gnatdist are, for the time being, exactly the same as for gnatmake. By default gnatdist outputs a configuration report and the actions performed. The switch -n allows gnatdist to skip the first stage of recompilation of the non-distributed application.
4. Start the distributed application by invoking the start-up shell script or Ada program (depending on the "pragma Starter" option, as will be shown later).

91

All configuration files should end with the '.cfg' suffix. There may be several configuration files for the same distributed application to permit the use of different distributed configurations tailored to the computing environment.

2. How Gnatdist Works

According to the GLADE User Manual (ACT Europe), the tool gnatdist works as following:

1. When building a distributed application, each compilation unit in the program is compiled into an object module (as in a non-distributed application). This is achieved by calling gnatmake on the sources of the various partitions. Using the -n switch skips this step.
2. Stubs are generated and compiled into object modules (a stub is the software that allows a partition running on machine A to communicate with a partition running on machine B). Several time stamp checks are performed to avoid useless recompilation.
3. Gnatdist performs a number of consistency checks. For instance, it checks to see that all packages marked as remote call interfaces (RCI) are mapped into partitions. It also checks to see that an RCI package is mapped onto only one partition.
4. Finally, the executables for each partition in the program are created. The code to launch partitions is embedded in the main partition, except if another option has been specified (pragma Starter). In this case, a shell script (or nothing) is generated to start the partition on the appropriate machines. This is especially useful when one wants to write client/server applications where the number of instances of the partition is unknown.

3. The Configuration Language

The configuration language is "Ada-like." Because of its simplicity, it is described by means of an example. As the capabilities of GLADE will evolve, so will this configuration language. Every keyword and construct defined in the configuration language has been used in the following sample configuration file, as shown in Figure 5.4. After having created the following configuration file you would typically type:

gnatdist my_config.cfg

If you wish to build only certain partitions, then list the partitions to build on the gnatdist command line as follows.

gnatdist my_config.cfg partition_2 partition_3

```
Configuration my_config is
    Partition_1: Partition := ( );
    Procedure Master_Procedure is in Partition_1;

    Partition_2, Partition_3: Partition;

    for Partition_2'Host use "foo.bar.com";

    function Best_Node (Partition_Name: String) return String;
    pragma Import (shell, Best_Node, "best-node");
    for Partition_3'Host use Best_Node;
    Partition_4: Partition := (RCI_B5);

    for Partition'Storage_Dir use "bin";

    procedure Another_Main;
    for Partition_3'Main use Another_Main;

    for Partition_4'Command_Line use "-v";

    pragma Starter (Method => Ada);

    pragma Boot_Server
        (Protocol_Name => "tcp". Protocol_Data => " 'hostname' : 'unused-port' ");

    pragma Version (False);

    Channel_1: Channel := (Partition_1, Partition_4);
    Channel_2: Channel := (Partition_2, Partition_3);

    for Channel_1'Filter use"ZIP";
    for Channel_2'Filter use "My_own_Filter;
    for Partition_3'Filter use "ZIP";

    pragma Registration_Filter ("Some_Filter");
begin
    Partition_2 :=  (RCI_B2, RCI_B4, Normal);
    Partition_3 :=  (RCI_B3);

end my_config;
```

Figure 5.4. Configuration File (ACT Europe)

According to the GLADE User Manual (ACT Europe), the following explains each line of code in file my_config.cfg, showed in Figure 5.4.

Configuration My_Config is

The name of the file prefix must be the same as the name of the configuration unit in this example "my_config.cfg." The file suffix must be .cfg. For a given distributed application you can have as many configuration files as you wish.

Partition_1: Partition := ();
procedure Master_Procedure is in Partition_1;

Partition 1 contains no RCI package. However, it will contain the main procedure of the distributed application, called "Master_Procedure" in this example. If the line *"procedure Master_Procedure is in Partition_1"* was missing Partition 1 would be completely empty. This is forbidden; a partition has to contain at least one library unit. Gnatdist produces an executable with the name of Master_Procedure, which will start the various partitions in the background on their host machines. The main partition is launched in the foreground. Note that by killing this main procedure the whole distributed application is halted (ACT Europe).

Partition_2, Partition_3: Partition;

for Partition2'Host use "foo.bar.com";

Declares two partitions called Partition_2 and Partition_3 and specifies the host on which to run partition 2.

function Best_Node (Partition_Name: String) return String;

pragma Import (Shell, Best_Node, "best-node");

for Partition_3'Host use Best_Node;

Use the value returned by a program to figure out at execution time the name of the host on which partition 3 should execute. For instance, execute the shell script "best-node" which takes the partition name as a parameter and returns a string giving the name of the machine on which partition_3 should be launched.

Partition_4: Partition := (RCI_B5);

Partition 4 contains one RCI package RCI_B5. No host is specified for this partition. The startup script will ask for it interactively when it is executed.

for Partition'Storage_Dir use "bin";

94

Specify the directory in which the executables in each partition will be stored and the directory in which all the partition executables will be stored, respectivelly. Default is the current directory.

procedure Another_Main;
for Partition_3'Main use Another_Main;

Specify the partition main subprogram to use in a given partition.

For Partition_4'Command_Line use "-v";

Specify the additional arguments to pass in the command line when a given partition is launched.

pragma Starter (Method => Ada);

Specify the kind of startup method you would like. There are 3 possibilities: Shell, Ada and None. Specifying "Shell" builds a shell script. All the partitions will be launched from a shell script. If "Ada" is chosen, then the main Ada procedure itself is used to launch the various partitions. If method "None" is chosen, then no launch method is used and you have to start each partition manually. If no starter is given, then an "Ada" starter will be used. In this example, Partition_2, Partition_3 and Partition_4 will be started from Partition_1 (i.e. from the Ada procedure Master_Procedure).

Pragma Boot_Server
(Protocol Name => "tcp", Protocol_data => "'hostname': 'unused-port'");

Specify the use of a particular boot server. It is especially useful when the default port 5555 used by GARLIC is already assigned.

pragma Version (False);

It is a bounded error to elaborate a partition of a distributed program containing a compilation unit that depends on a different version of the declaration of RCI library unit than that included in the partition to which the RCI library was assigned. When the pragma Version is set to false, no consistency check is performed (ACT Europe).

Channel_1: Channel := (Partition_1, Partition_4);
Channel_2: Channel := (Partition_2, Partition_3);

for Channel_1'Filter use "ZIP";

Declare two channels (other channels between partitions remain unknown), and use transparent compression/decompression for the argument and results of any remote calls on channel "Channel_1," i.e. between "Partition_1" and "Partition_4."

95

for Channel_2'Filter use "My_Own_Filter";

for Partition_3'Filter use "ZIP";

Use filter "My_Own_Filter" on "Channel_2." This filter must be implemented in a package "System.Garlic.Filters.My_Own_Filter." For all data exchanged with "Partition_3", except "Partition_2", use the filter "ZIP" (i.e. for both arriving remote calls as well as for calls made by this partition). Only for calls on "Channel_2" (i.e. for communication between "Partition_2" and "Partition_3") is the filter "My_Own_Filter" used.

pragma Registration_Filter ("Some_Filter");

"Some_Filter" will be used to exchange a filter's parameters between two partitions. "Some_Filter" itself must be an algorithm that does not need its own parameters to be filtered again. On all other channels (i.e. for remote calls between partitions where no channel was declared), filtering is not used.

Begin

The configuration body is optional. You may have fully described your configuration in the declaration part.

Partition_2 := (RCI_B2, RCI_B4, Normal);

Partition_3 := (RCI_B3);

Partition 2 contains two RCI packages RCI_B2 and RCI_B4 and a normal package. A normal package is not categorized. Partition 3 contains one RCI package RCI_B3.

a.   Remote Shell

To start a partition, the main partition executable executes a remote shell. Thus, you must ensure that you are authorized to execute a remote shell on the remote machine. In this case, a first step would be to add into your $(HOME)/.rhosts file a line like:

<remote-machine> <your-username>

If you are not authorized at all, you can bypass this problem. According to GLADE User Manual (ACT Europe), all you have to do is:

1. Open a session on each machine listed on your configuration file.
2. If MAIN_PART is the partition that includes the main procedure and if you want to start MAIN_PART on host MAIN_HOST, then:

a) Choose a TCP port number PORT_NUM (gnatdist default is 5555 when using a shell starter, randomly chosen when using an Ada starter).

b) Then for each partition PART, start manually the corresponding executable on the corresponding host as follows:

% PART [--nolaunch] [--slave] --boot_server tcp://MAIN_HOST: PORT_NUM

The --nolaunch parameter must be included for the main partition, it means that this partition is not in charge of launching others. The --slave parameter must be included for other partitions, meaning that in no case the name server is located on them.

3. If you want to kill the distributed application before it terminates, kill MAIN_PART.

b.   *Filtering*

GLADE contains a transparent extensible filtering mechanism allowing the user to define various data transformations to be performed on the arguments and return the values of remote calls. One possible application would be to compress all data before sending it and then decompress it on the receiving partition. As default, no filtering is performed by GLADE, but the compression filter is available. Therefore, you can configure your distributed application in order to use this filter.

The configuration language not only knows about partitions, it also knows about the connections between them. Such a connection is called "Channel" and represents a bi-directional link between two partitions. In order to define filtering, one must first declare the channels between the partitions of an application:

A_Channel: Channel := (Partition_1, Partition_2);

This gives the link between partitions "Partition_1" and "Partition_2" the name "A_Channel". It is not possible to declare more than one channel between the same two partitions.

Now that this channel is known, the data transformation that is to be applied on all data sent through it can be defined:

For A_Channel'Filter use "ZIP";

This specifies that all data sent over this channel should be transformed by the filter named "ZIP." There should be a filter with this name, implemented in the package "System.Garlic.Filters.Zip."

Some filtering algorithms require that some parameters must be sent to the receiver first to enable it to correctly un-filter the data. In this case, it may be necessary

97

to filter these parameters again. For such purposes, it is possible to install a global filter for all partitions, which then will be used to filter the parameters of other filters. This filter is called the "registration filter." It can be set by a pragma, as shown below.

pragma Registration_Filter ("Filter_Name");

It may also be useful to specify that a partition uses a certain filter for all remote calls, regardless of the channel (i.e. regardless of the partition that will receive the remote call). This can be specified using the attribute 'Filter on a partition:

for Partition_1'Filter use "ZIP";

or even

for Partition'Filter use "ZIP";

The latter set the default filter for all partitions of the application, the former only sets the default filter for the partition "Partition_1." We may specify the attribute 'Filter using either the latter or the former specification.

Gnatdist takes care of consistency checking of a filter definition. By default, no filtering is done. Filtering is only active if specified explicitly in the configuration file (ACT Europe).

As has been briefly mentioned above, a filter with a name "NAME" must be implemented in a package called "System.Garlic.Filters.Name." You may write your own filters, which must implement their filtering of data in the primitive operations of a type derived from the type "System.Garlic.Filter_Type." Your filter package must then register an instance of your newly derived type with GLADE by calling "System.Garlic.Filters.Register." From that on, your filter is ready to be used.

4. Foreign Code

GLADE allows the user to make calls to a foreign code, for example a C routine, from inside a partition in a distributed program. In this case, the foreign routine should be encapsulated by an Ada procedure or function. Also, foreign library units, as well as the compilation units needed by those library units, should be explicitly assigned using pragma Linker_Options. Figure 5.5 below illustrates the encapsulation of a C unit called cooler_display.c.

```
package Gui_Pkg is

        -- explictly assign the foreign libraries
        pragma Linker_Options ("cooler_display.c");
        pragma Linker_Options ("other libraries to link with");

        -- encapsulate the foreign routines
        procedure InitializePanels;
        pragma Import (C, InitializePanels, "InitializePanels");

        procedure Display (i : in integer);
        pragma Import (C, Display, "Cooler_Display");

end Gui_Pkg;
```

Figure 5.5. Encapsulation of a C Routine

5.   Debugging

GLADE has a facility for trace/replay based debugging. If trace mode is turned on, GLADE will record all messages received by a partition into a trace file. The trace file can then be used to replay the execution of the partition in isolation.

To get a partition to generate a trace file, it has to be passed the command line argument "--trace." This is most easily done by using the "for Partition'Command_Line use…" construct described above in the configuration file to add "--trace" to the command lines of the partitions whose executions are to be replayed. When the application has been built, starting it using the starter as usual will result in a trace file being generated.

By default, the file name of the trace file is the name of the partition's executable with a trailing ".trace." This can be changed with the "--trace_file othername" command line argument. Note that since the remote partitions are launched with rsh under Unix, the current directory during execution will be the user's home directory. This is no problem when using the default trace file name, because the executable's name will include the absolute path. When using the "--trace_file" option, on the other hand, if you do not want the trace file to be created/read in the home directory, the absolute path will have to be included in the desired name. (ACT Europe)

To replay a partition whose execution has been previously traced, the command line argument "--replay" is required. In addition, the special boot server location

"replay://" has to be specified, i.e. by using the "--boot_server replay://" command line argument. For example, to replay a traced execution of a partition that has an executable named PAR you would start it with the command:

% PART [--nolaunch] [--slave] --replay --boot_server replay://

possibly under the control of a debugger, such as gdb.

Since the exact contents of the messages received is recorded, differences in input from external sources (such as standard input) during replay will most likely give unexpected results. Also, replay of applications whose behavior is inherently non-deterministic will be problematic. It is important that the same executable is used for replay as when the trace file was generated, otherwise strange behavior can be expected. (ACT Europe)

6.   Restrictions

Currently the following restrictions apply to GLADE:

1. Static remote procedures, asynchronous remote procedures, remote access to class wide types and asynchronous transfer of control with remote procedures are implemented;
2. Remote access to subprogram have not yet been implemented;
3. Pragma ALL_Calls_Remote, shared passive packages and generic RCI packages has not yet been implemented;
4. Language-defined exceptions propagate well through different partitions.

For the time being, gnatdist is only able to build distributed applications for a pool of homogeneous or heterogeneous machines using TCP/IP as a common network protocol (ACT Europe).

Besides the restrictions above, our experiments show that there is a problem when we try to execute an I/O command in a remote partition when using method Ada to automatically launch the partition. This problem seems to happen because the partition is launched and executed in the background of the remote machine, therefore we cannot see any I/O command. This is especially bad when we assign a user interface unit to the remote partition. For example, if a distributed program fails to initialize a panel in a remote machine, then the whole program will be blocked when trying to send the data to be displayed on the panel.

C. THE ADA 95 DISTRIBUTED ARCHITECTURE

Figure 5.6 below illustrates the Ada 95 distributed architecture. It shows a non-distributed application with three modules called A, B and C. Module A makes calls to modules B and C, and passes some arguments, which are received by B and C and used in their computation.
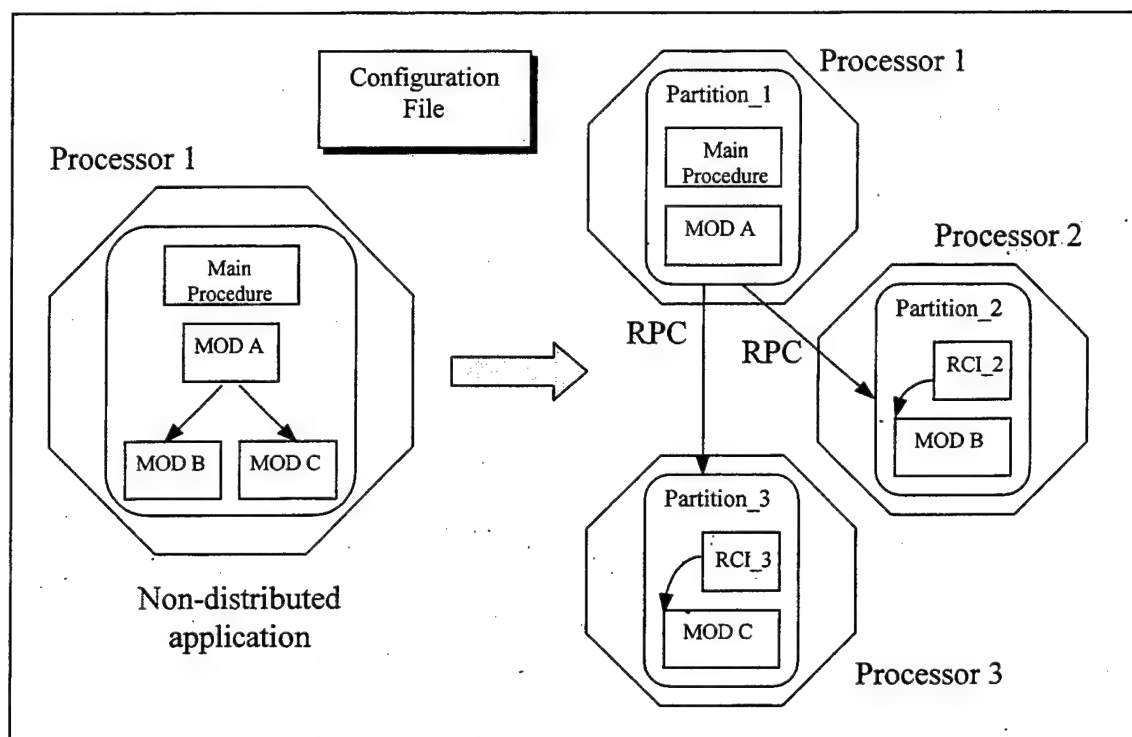
Figure 5.6. The Ada 95 Distributed Architecture

After testing the non-distributed application, the user creates a configuration file to configure the partitions of the program. In this case, we have three partitions: Partition_1, Partition_2 and Partition_3. As we can see in Figure 5.6, module A is assigned to Partition_1, module B to Partition_2 and module C to Partition_3. The partitions are mapped to processor 1, processor 2, and processor 3, respectively. Partition_1 is the main partition. It contains the main procedure, which will start the various partitions.

To receive the remote calls, the partitions Partition_2 and Partition_3 contain a categorized unit called RCI_2 and RCI_3, respectively, which have the categorization pragma Remote_Call_Interface. The interesting point of this distributed architecture is that since GLADE supports the asynchronous mode of communication, it is possible to have modules A, B and C working in parallel, but on different machines.

101

VI.   PROTOTYPE OF THE SOFTWARE ARCHITECTURE

In this chapter, we discuss the characteristics of the Computer Aided Prototyping System (CAPS) and explain the current CAPS architecture for the uniprocessor implementation. Then, we propose an architecture for a distributed implementation based on the GNAT Library for Ada Distributed Execution (GLADE). As we saw in the previous chapter, with GLADE it is possible to create a uniform Ada 95 distributed system without having to interface to any low-level communication layer. By combining the distributed and real-time capabilities of Ada 95, it is possible to design prototypes which meet real-time constraints in a distributed environment.

A.   THE CURRENT UNIPROCESSOR ARCHITECTURE

CAPS uses the prototype system description language (PSDL) to integrate a set of tools, including an execution support system and a syntax-directed editor with graphic capabilities, which are linked together by a user interface. PSDL provides two kinds of basic building blocks for prototypes: data types and operators. According to Luqi (1988), these constructs are sufficient to specify a prototype's design and structure.

Software Systems are modeled as networks of operators communicating via data streams. The networks are represented as dataflow diagrams with a bubble for each operator and an arrow for each data stream. PSDL provides graphical notation for dataflow diagrams enhanced with nonprocedural control and timing constraints. Control constraints manage the output and the firing of an operator, while timing constraints define the execution time, periodicity and deadline of a time-critical operator. Figure 6.1 shows an example, the Temperature Control PSDL graph.
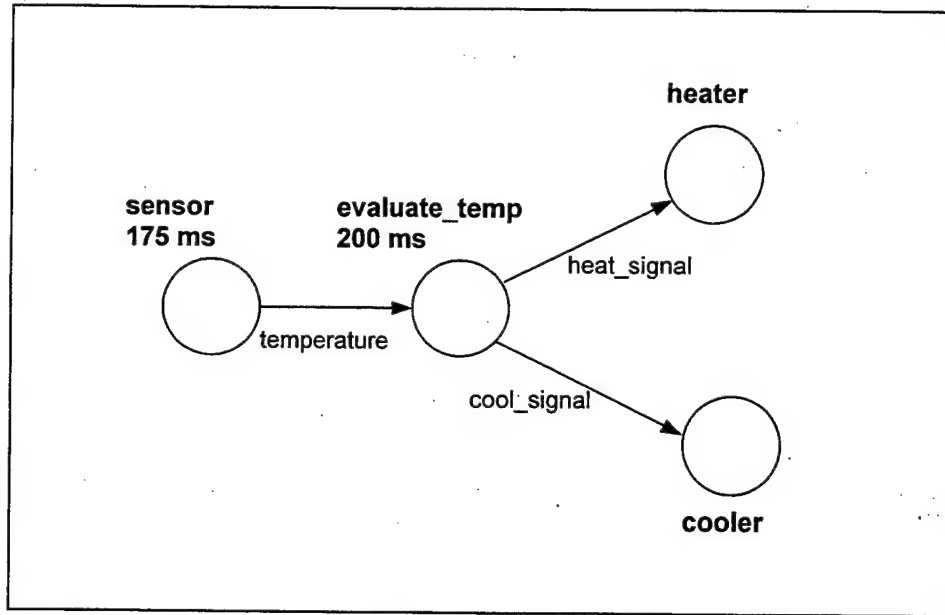
Figure 6.1. Temperature Control PSDL Graph

The PSDL execution-support system contains a translator, a static scheduler, and a dynamic scheduler. The translator automatically generates code. Its main functions are to implement data streams and control constraints. The static scheduler allocates time slots for operators with real-time constraints, that is, time-critical operators. If the allocation succeeds, all operators are guaranteed to meet their deadlines even with worst-case execution times. The dynamic scheduler invokes operators without real-time constraints in the time slots not used by time-critical operators.

More specifically, the translator converts the PSDL program defined by the user into compilable Ada units. During this process, it creates the following major packages: exceptions, instantiations, timers, streams, and drivers, all preceded by the name of the prototype followed by an underscore. Ultimately, each of these will become part of the prototype supervisory Ada program (Cordeiro, 1995). A partial view of the supervisory program for the Temperature Control prototype is shown in Figure 6.2.

```
Package TEMP_CONTROLLER_EXCEPTIONS is
    -- PSDL exception type declarations
    type PSDL_EXCEPTION is (UNDECLARED_ADA_EXCEPTION);
end TEMP_CONTROLLER_EXCEPTIONS;

package TEMP_CONTROLLER_INSTANTIATIONS is
    -- Ada Generic package instantiations
end TEMP_CONTROLLER_INSTANTIATIONS;

-- with/use clauses for CAPS library packages.
with PSDL_STREAMS; use PSDL_STREAMS;
package TEMP_CONTROLLER_STREAMS is
    -- local streams instantiations
    package DS_TEMPERATURE_EVALUATE_TEMP is new
        PSDL_STREAMS.SAMPLED_BUFFER(FLOAT);
    package DS_HEAT_SIGNAL_HEATER is new
        PSDL_STREAMS.SAMPLED_BUFFER(BOOLEAN);
    package DS_COOL_SIGNAL_COOLER is new
        PSDL_STREAMS.SAMPLED_BUFFER(BOOLEAN);
    -- State stream instantiations
end TEMP_CONTROLLER_STREAMS;

package TEMP_CONTROLLER_DRIVERS is
    procedure SENSOR_DRIVER;
    procedure HEATER_DRIVER;
    procedure COOLER_DRIVER;
    procedure EVALUATE_TEMP_DRIVER;
end TEMP_CONTROLLER_DRIVERS;
```

Figure 6.2. Partial View of Temp_Controller.a

The first three of these packages contain all of the user declared exceptions, generic packages and timer instantiations defined in the PSDL program. The package streams contain the instantiations of all the streams used by the prototype, which are implemented as Ada generic packages that contain protected buffer objects with operations READ, WRITE and NEW_ELEMENT. The package PSDL_STREAMS contains all stream types supported by PSDL. Finally, the package drivers contain all of the data declarations, the data trigger checks that control whether a stream should be read, the execution trigger checks that decide whether an operator should be fired, and the output guard checks, which decide whether a computed result should be written to the output streams (see Section A.2 of Chapter III for details).

In addition to these packages generated by the translator, there are another two packages generated by the scheduler. When consolidated by one of the CAPS scripts, they will form a *prototype supervisory program*, receiving the name of the prototype followed by an ".a" extension, which stands for Ada program (Cordeiro, 1995). Figure 6.3 shows the structure of the CAPS supervisory program.

```
Exceptions Declarations
Generic Instantiations
Timer Instantiations
Data Streams Instantiations
Operator Drivers
```
CAPS
Support Packages

```
while true loop
    call non-time-critical operator drivers;
end loop;
```
Dynamic Schedule

```
while true loop
    call time-critical operator drivers;
end loop;
```
Static Schedule

```
procedure prototype_name is
begin
    init_hardware_model;
    start_static_schedule;
    start_dynamic_schedule
end prototype_name;
```
Main Program

Figure 6.3. CAPS Supervisory Program Structure

The scheduler generates the *static scheduler task* that is responsible for calling all time-critical operators, according to the static schedule. The time-critical operators will be called in a non-preemptive way, so that each instance of an operator will execute to completion. The scheduler also generates the *dynamic scheduler task* that is responsible

for calling all non-time-critical operators of the prototype. They run in a pre-defined order established by the dynamic schedule whenever there is idle time in the static schedule. The *dynamic scheduler task* has lower priority than the *static scheduler task*, so non-time-critical operators can be preempted by time-critical ones.

B. THE PROPOSED DISTRIBUTED ARCHITECTURE

One of our objectives when designing this new distributed architecture was to minimize the changes in the current CAPS architecture. CAPS models software systems as networks of operators communicating via data streams, which is by itself very close to the concept of distributed systems. Hence, the CAPS model is very suitable to be implemented in a distributed architecture. It does not matter where the operators are located in the system, only that the system should be able to transmit data in an appropriate way and to handle the real-time constraints.

According to Cordeiro (1995), in the uniprocessor case, the translator had no information about the output of the scheduler. For the distributed case, however, this information is crucial, since it will have to generate different Ada units for each of the processors involved in the prototype. Once the scheduler has defined the different partitions and the operators that belong to each partition, the translator will have to be called, so that it can generate as many supervisory files as the number of partitions.

In this implementation, due to a restriction in GLADE that does not recognize the extension ".a," we split the packages in the supervisory files into distinct modules, and use a naming convention for the different files the name of the prototype followed by the partition number and the name of the package, e.g., autopilot_1_drivers, autopilot_1_static_schedulers, and so on. Note that in GLADE the different files should receive the name of the corresponding package. Also, we recommend that only lower case letters be used in the file names to avoid problems with the UNIX file system.

For example, the prototype for an Autopilot is illustrated in Figure 6.4 and the corresponding global precedence graph in Figure 6.5 below. Note that the Autopilot PSDL graph contains cycles and we will show in Section B of Chapter VII that such prototypes will work in the distributed environment as long as we assign the operators in each cycle to the same processor.

107
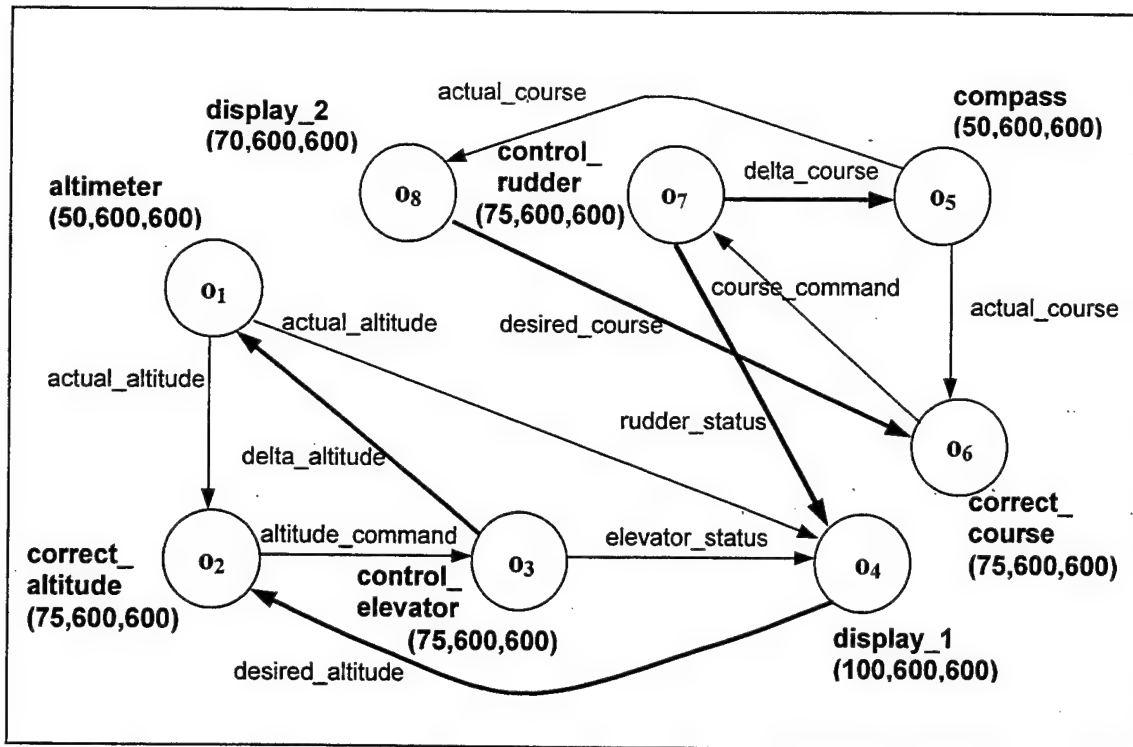
Figure 6.4. Autopilot Prototype

Applying the distributed scheduling algorithm we proposed in Chapter IV to the Autopilot prototype, the scheduler first defines the number of available processors (in this case two processors) and then allocates each operator in the prototype to a processor. The output of the distributed scheduling algorithm with a *max_load_factor* of 0.7 is illustrated in Figure 6.6.

108
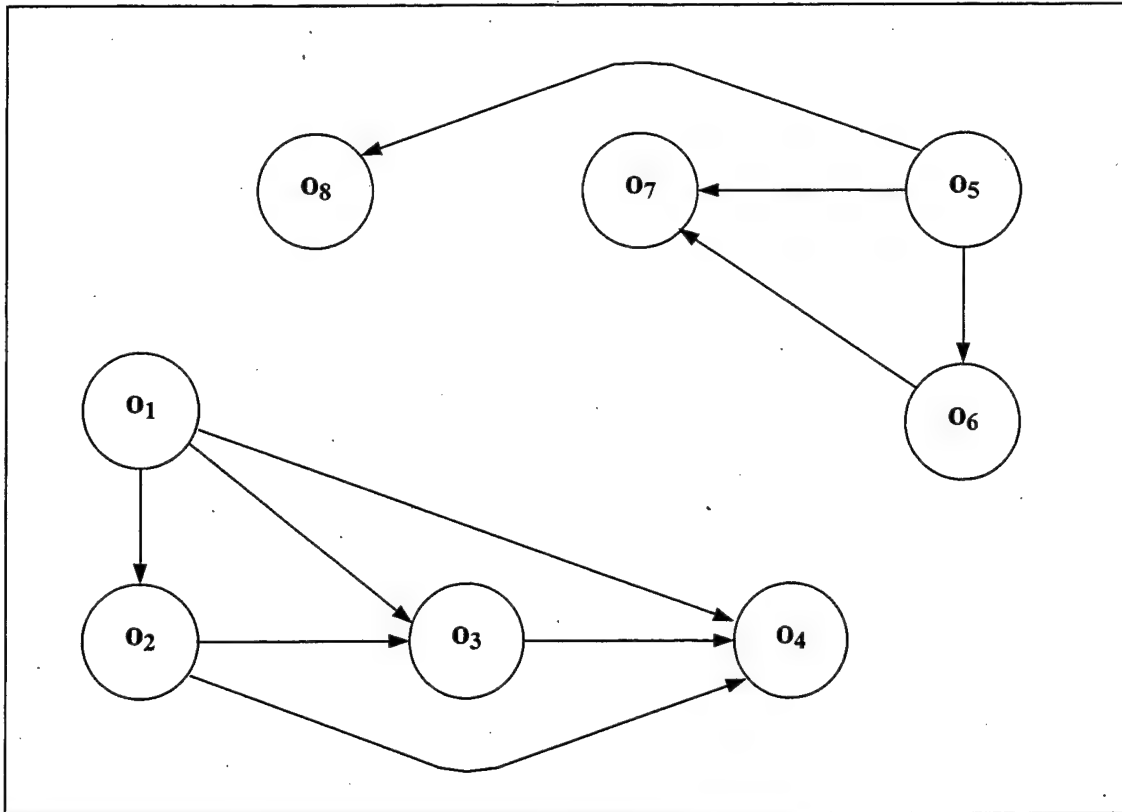
Figure 6.5. The Autopilot Global Precedence Graph
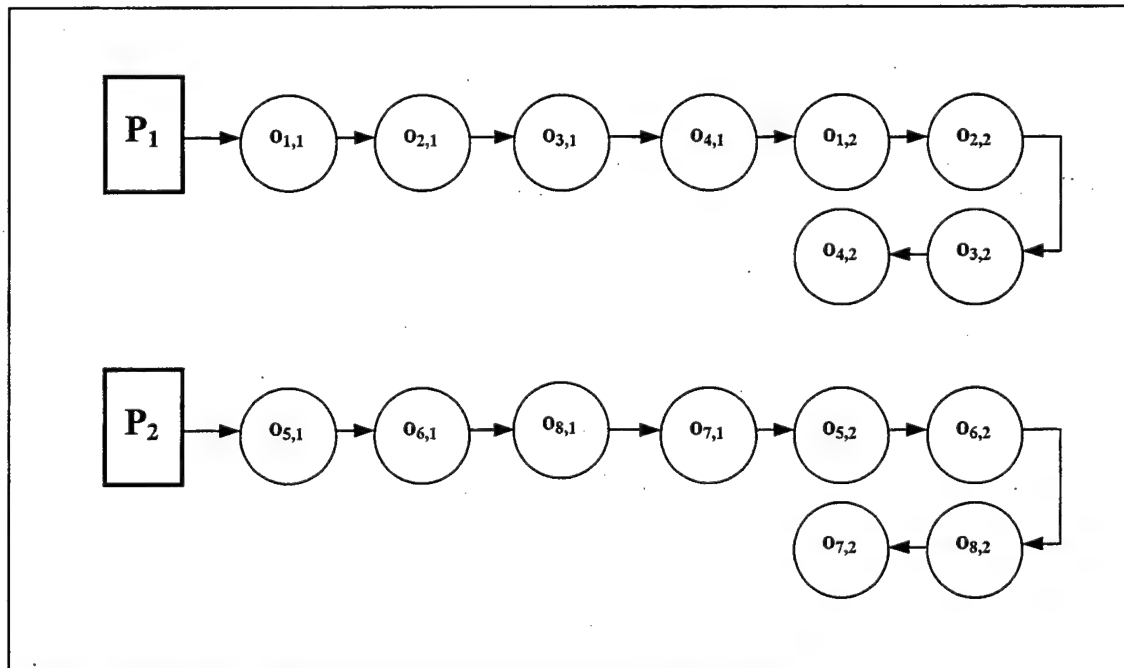
Figure 6.6. Output of the Distributed Scheduling Algorithm

109

According to Cordeiro (1995), the following information should be passed by the scheduler to the translator, so it can perform its job.

1. Number of partitions and a list with the operator name belonging to each partition
2. Mapping from partitions to processors

As we can see in Figure 6.6, the output of the scheduler contains the necessary information to satisfy the first item in the list above. To satisfy the second item, it is sufficient to provide the translator with the name of the available processors in the network. We recommend that this information come from the CAPS user interface. Once all this information is available to the translator, it should generate all necessary packages for each partition, exactly as it did for the uniprocessor implementation, except for the following differences that we explain below.

1.   Package Remote_Streams

For the uniprocessor case, the package PSDL_Streams contains all of the types of streams available in PSDL. However, for the distributed case where operators communicate via network, we need a special kind of stream, i.e., the remote stream. Remote streams do not need to be represented in the prototype and can be transparent to the user. They are implemented by a package called remote_streams that contains the categorization pragma Remote_Call_Interface. This package is used as an interface for Remote Procedure Calls (RPC) between partitions. For each partition that receives one or more interprocessor communications, the translator should generate a corresponding package remote_streams that handles all incoming communications. Following the naming convention we are using here, this file should be named according to the partition to which it belongs, e.g., the autopilot_1_Remote_Streams and the autopilot_2_Remote_Streams shown in Figure 6.9.

Because GLADE currently does not support generic remote call interface packages and, the declaration of pragma Remote_Call_Interface should precede any other declaration in the package, the remote_streams can be neither instantiated nor part of package PSDL_Streams like the other types of streams. Note that an operator calls package remote_streams only when it needs to write to a stream that is external to its partition. It works like a bridge connecting two partitions. After receiving the data from a remote producer operator the corresponding write procedure will be invoked to write

the data in the corresponding output stream (refer to Figure 6.7 for the specification of the remote_streams package).

In this implementation we consider that all operators read from local streams. It is very important to notice that here we assume that a stream is instantiated in the same processor or partition of its consumer operator. Therefore, it is irrelevant where the data comes from.

Finally, for this implementation to conform with the distributed scheduling model without synchronization, the translator should be smart enough to change any data flow stream that receives data from the network into sampled stream, so that no overflow or underflow exceptions will be raised for interprocessor communications. Figure 6.7 shows the specification of the new package remote_streams for partition 1 of the Autopilot distributed prototype.

```
with RUDDER_STATUS_TYPE_PKG; use RUDDER_STATUS_TYPE_PKG;
package Autopilot_1_REMOTE_STREAMS is

    -- use the categorization pragma to create an interface for RPCs
    pragma Remote_Call_Interface;

    -- write the rudder status data to the rudder_status  stream
    procedure Write_Rudder_Status_Display_1
                            (Rudder_Status : in RUDDER_STATUS_TYPE);

    -- define the IPCs as asynchronous RPCs
    pragma Asynchronous (Write_Rudder_Status_Display_1);

end Autopilot_1_REMOTE_STREAMS;
```

Figure 6.7. Specification of Package Autopilot_1_REMOTE_STREAMS

2.   The New Package Drivers

The new package drivers should contain only the driver procedures related to the operators belonging to that partition. All reads will be local streams. However, if it is necessary to write to an external operator, the translator should replace the usual write operation by the corresponding write procedure of the remote stream package

3.   The Tasks Static Schedule and Dynamic Schedule

The output of the distributed scheduling algorithm contains the schedule for each one of the processors or partitions in the distributed system. Therefore, the new task

111

Static Schedule should call only the time-critical operators belonging to that partition. Likewise, the task Dynamic Schedule should call only those operators that are not time-critical, and that belong to that partition. Note that this implementation conforms to the distributed scheduling model without synchronization, where, ideally, sets of communicating processes would run independently in each processor.

4. The Configuration File

The process of mapping the partitions of the program to the nodes in a distributed system is called configuring the partitions. With GLADE, a configuration file (as we saw in Chapter V) should be created to configure the partitions. So, in addition to generating the six major packages (exceptions, instantiations, timers, streams, remote_streams and drivers), the translator has the new job of generating the configuration file for the distributed implementation. Figure 6.8 illustrates the configuration file for the Autopilot distributed prototype.

```
Configuration Autopilot is
    pragma version (False);
    pragma Starter (None);
    -- The partitions will be launched manually

    pragma Boot_Server ("tcp", "sun53 : 3333");

    partition1 : Partition := (PSDL_Streams, autopilot_1_exceptions,
                               autopilot_1_instantiations, autopilot_1_timers,
                               correct_altitude_PKG, altimeter_PKG, gui_PKG,
                               display_1_PKG, altitude_command_type_PKG,
                               control_elevator_PKG, rudder_status_type_PKG,
                               elevator_status_type_PKG, course_command_type_PKG,
                               autopilot_1_drivers, autopilot_1_dynamic_schedulers,
                               autopilot_1_static_ schedulers, autopilot_1_start_drivers,
                               autopilot_1_streams, autopilot_1_remote_streams);

    partition2 : Partition := (PSDL_Streams, autopilot_2_exceptions,
                               autopilot_2_timers, rudder_status_type_PKG,
                               autopilot_2_instantiations, compass_PKG,
                               correct_course_PKG, control_rudder_PKG,
                               course_command_type_PKG, display_2_PKG,
                               autopilot_2_drivers, autopilot_2_dynamic_schedulers,
                               autopilot_2_static_ schedulers, autopilot_2_start_drivers,
                               autopilot_2_streams, gui_pkg);

    procedure Start_Autopilot is in partition1;
    -- procedure Start_Autopilot will start the execution of the Autopilot execution

    -- procedure Start_Partition2 is the driver procedure for partition2
    procedure Start_Partition2;
    for partition2'Main use Start_Partition2;

    channel_1 : Channel := (partition1, partition2);

end Autopilot;
```

Figure 6.8. The Configuration File for the Distributed Autopilot Prototype

5.   The Architecture of the Distributed Implementation

Figure 6.9 illustrates the distributed implementation of the Autopilot prototype shown above. Note that when an operator writes to an external operator, the driver procedure calls package Remote_Streams instead of the normal Streams.

113
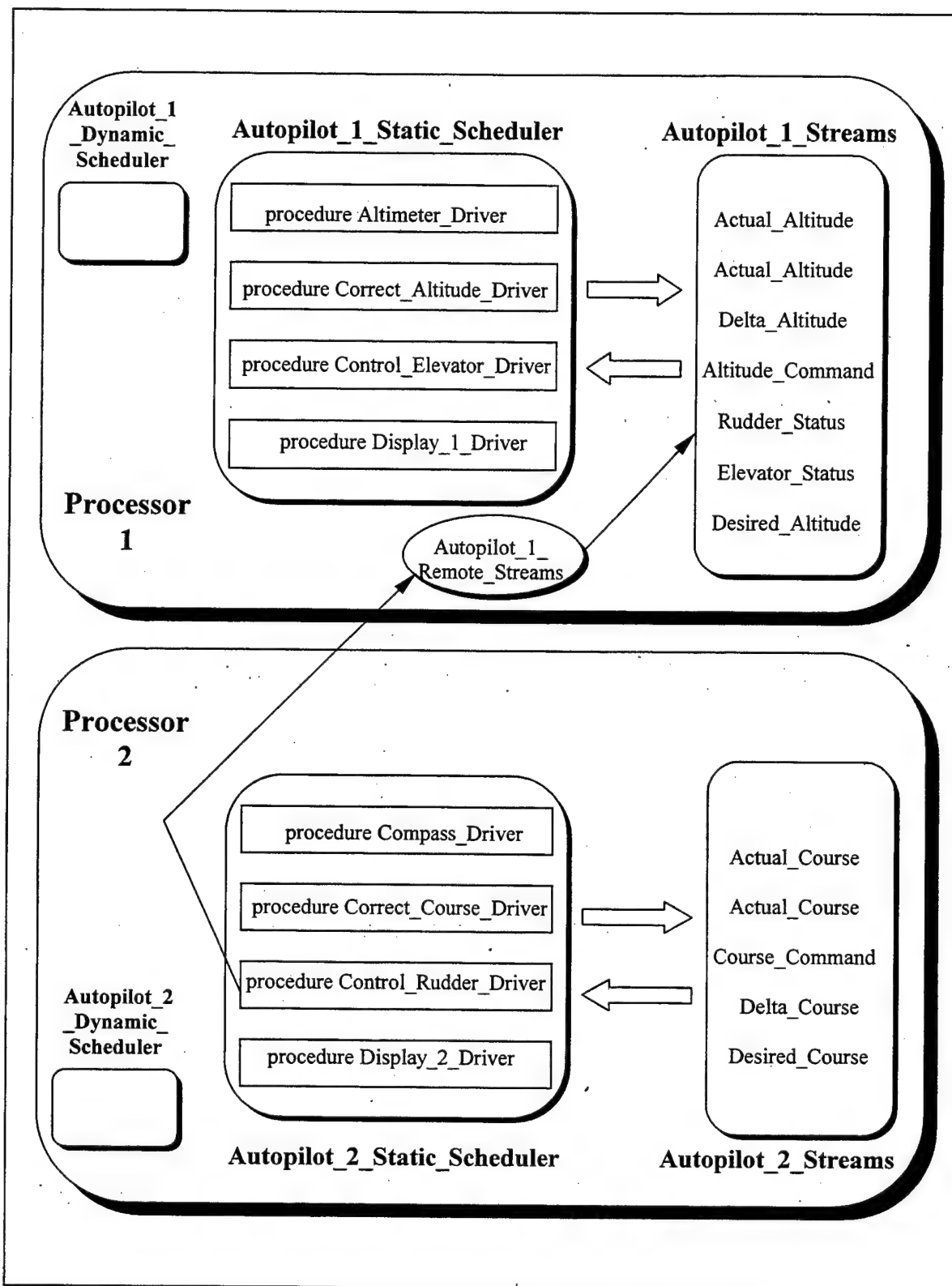
Figure 6.9. Architecture of the Distributed Autopilot Implementation

VII. CONCLUSIONS AND RESULTS

A. SUMMARY OF THE THESIS

This thesis deals with the area of computer-aided real-time distributed embedded systems development. The Prototyping System Description Language (PSDL) is a language designed for clarifying the requirements of complex embedded systems. It simplifies the design of systems with real-time constraints by presenting a high level description in terms of networks of independent operators to the designer. PSDL is based on a computational model containing operators that communicate via data streams, where each stream carries values of a fixed abstract data type. The PSDL computational model is represented as an augmented graph.

One of the most important properties of any real-time system is that its behavior be predictable. It should be clear at design time that the system can meet its deadlines, even in the worst case condition. In order to satisfy this requirement static scheduling is done before the system starts operating. The input consists of a list of all operators and their time constraints. The output consists of an assignment of operators to the available processors, and for each processor, a static schedule giving the order and times the operators are to be executed.

For the uniprocessor case, the scheduler simply analyzes the PSDL graph and the timing constraints of each operator to build the schedule. For the distributed case, however, the scheduler has, also, to allocate each operator to a processor and take communication into account. The problem of scheduling the same operators onto a set of processors changes due to the delay itroduced by the communication network.

The motivation to build the distributed schedule is based on the fact that concurrent processing is essential because the only way to make some real-time constraints feasible is to use multiple processors.

In Chapter IV, we propose a distributed scheduling algorithm to deal with the allocation and scheduling problem. The distributed scheduling algorithm is based on the distributed scheduling model with no explicit synchronization (Cordeiro, 1995), where each set of operators allocated to a particular processor can be treated as a totally independent set.

115

The technique we propose uses ordered doubly linked list structures to represent processors and the communication network. Each element of a processor represents an operator instance. For the communication network list, each element represents an incoming communication arriving at the corresponding processor.

Basically, the distributed scheduling algorithm allocates the operators defined in the PSDL graph to the available processors. Then, it searches the edges existing in the graph and checks if there is any interprocessor communication due to the allocation of operators to different processors. If necessary, the algorithm updates the start time and completion time of the operators affected by the delay introduced by that interprocessor communication. Finally, the algorithm verifies if the schedule is feasible, that is, if the completion times of all operator instances are less than their deadlines, and the precedence constraints are satisfied.

1. Implementation

In this thesis, we investigate the capabilities of distributed real-time systems support in Ada 95. The most significant objective is the design and development of an Ada 95 software architecture for distributed real-time embedded systems and automatic generation tools for such architecture.

The Ada 95 Distributed Systems annex (Ada, 1995) defines facilities for supporting the implementation of distributed systems using multiple partitions working cooperatively as part of a single Ada program. In Chapter V we present GLADE (GNAT Library for Ada Distributed Execution), an implementation of the Distributed Systems Annex for the GNAT compiler. By combining the distributed and real-time capabilities of Ada 95, it is possible to design systems which meet real-time constraints.

In Chapter VI, we discuss the characteristics of CAPS (Computer Aided Prototyping Systems), which uses the PSDL language to integrate a set of tools, including an execution support system and editor with graphic capabilities, to prototype large and complex real-time systems. We further propose a distributed architecture for the CAPS generated control code.

The CAPS execution support contains a translator, a static scheduler and a dynamic scheduler. The static and dynamic schedulers are generated by the CAPS scheduler. The translator converts the PSDL program defined by the user into compilable

116

Ada units, the static scheduler invokes time-critical operators and the dynamic scheduler invokes operators without real-time constraints.

In the uniprocessor case the translator does not need any information about the output of scheduler. For the distributed case, however, the translator needs this information in order to generate different Ada units for each of the processors involved in the prototype. The following information should be passed to the translator.

1. Number of partitions and a list with the operator name belonging to each partition
2. Mapping from partitions to processors

The output of the distributed scheduling algorithm we proposed in Chapter IV can provide the information about the number of partitions and the operators belonging to each partition (see Figure 6.6). The name of the processors available on the network should be inserted directly by the user through the CAPS interface.

Once these informations are available to the translator, it should generate all necessary packages for each partition. The major difference is that besides the usual packages, the translator must generate two other units: the configuration file and the package remote_streams. The configuration file is created to configure the partitions (see Figure 6.8), that is, to map partitions to the available processors.

Package remote_streams is a unit that has the declaration of the categorization pragma Remote_Call_Interface (see Figure 6.7), and should be created in each partition that receives an interprocessor communication. When an operator needs to write data to a stream that is external to its partition, and only in this case, it calls package remote_streams. After receiving the data from the remote producer operator, the corresponding write procedure will be invoked to write the data in the corresponding output stream.

B.   EXPERIMENTAL RESULTS

We have conducted two different experiments in wich the capabilities of distributed real-time systems support in Ada 95 was evaluated. The first experiment investigates the latency for transmission of a data value from processor i to processor j in a homogeneous network (refer to Section A.6 in Chapter III). In the experiment we calculated the average of the latency for transmission of different data types, using Ada native and user defined types. Figure 7.1 shows the results of the experiment.

117

Data Type	Size (bits)	Latency (seconds)
Integer	16	1.792744
Float	32	1.815710
Array (size 100) of integer	100×16	1.785339
Array (size 100) of float	100×32	1.784241
Record w/ two arrays (size 100)	$100 \times 16 + 100 \times 32$	1.785088
Array (size 1000) of integer	1000×16	1.792962
Array (size 1000) of float	1000×32	1.7855356
Record w/ two arrays (size 1000)	$1000 \times 16 + 1000 \times 32$	1.792626

Figure 7.1. Latency for Interprocessor Communications

To perform the experiment, we used GLADE release 1.03p to build a distributed application running on top of Sun OS release 4.1.3 on SPARC Station 2 machines, connected by a 100Mbps FDDI network. As we can see in Figure 7.1, the latencies for interprocessor communication are too big. In addition, increasing the size of the data does not increase the latency. It seems that the problem is the delay introduced by the communication subsystem and not by the network itself. Further experiments using other platforms are needed, because few practical real-time systems can meet their deadlines with such big communication delays. According to the GLADE User Manual (ACT Europe), GLADE release 1.03p is fully supported by Sparc/Solaris, PC/Linux and Alpha/Decunix.

In the second experiment, several prototypes were implemented based on the proposed CAPS distributed architecture implementation discussed in Chapter VI. The results indicate that we cannot apply the distributed scheduling algorithm proposed in Chapter IV to any PSDL graph. If the user wants to build a prototype that works under the distributed implementation, it should be carefully designed to fit the model. The reason is that we cannot have cycles crossing the network. Cycles are dangerous because they require explicit synchronization, which would make scheduling impossible.

In the uniprocessor case, cycles can be broken by state streams because there are no implicit precedence constraints associated with state streams. Although state streams do not imply any precedence constraints, we still have to consider the interprocessor

118

communication delay to send data and to receive the feedback loop in the distributed case.

In one of our experiments, we applied the algorithm described in Chapter IV to the Autopilot prototype (designed for the uniprocessor model), and implemented a distributed prototyping according to the architecture proposed in Chapter VI. The initial version of the distributed Autopilot prototype contained cycles crossing the network. The prototype worked poorly and was very unstable, could not always keep the desired course and altitude. We redesigned the original prototype to fit the distributed model (see Figure 6.4) and the new version of the Autopilot prototype worked well. Except for the long delay to display the information about the rudder status due to the large interprocessor communication between operator control_rudder and display_1. The source code for this new implementation can be found in the Appendices.

C. CONCLUSIONS AND FUTURE WORK

This thesis proposes a distributed scheduling algorithm and a distributed implementation for the current CAPS architecture based on GLADE. The practical effeceiveness of the proposed distributed scheduling algorithm can only be tested in realistic settings. Thus, the idea proposed here should be implemented and tested on a large sample of prototypes. Because we already know that there is no optimal solution for the distributed scheduling problem, further research is necessary to improve the algorithm and find new ideas to deal with the problem.

The new CAPS architecture was implemented and applied to several example prototypes. Preliminary results from current experiments showed that it is possible to build distributed real-time embedded systems under the distributed scheduling model, where sets of tasks run independently on each processor, using GLADE. We also learned about limitations of this new technology. One limiting factor is the presence of cycles crossing the network. Prototypes intended to be used in a distributed environment should be specially designed, so that cycles are confined within each processor.

Another limiting factor is the large latency for interprocessor communications. Our experiments showed that the communication subsystem, not the network itself, is responsible for increasing the latency. Further experiments using other platforms should be conducted in order to find better results.

119

1. Possible CAPS Modifications

As a result of the ideas proposed in this thesis, few modifications to CAPS are required. The most significant modification is related to the translator. In the distributed scheduling model the translator should receive information about the number of partitions and operators belonging to each partition from the scheduler, in order to generate the necessary files for each processor. In addition, it should receive information about the names of the available processors in the network from the CAPS user interface, in order to map each partition to a processor. Hence, the CAPS user interface should be modified to allow user to enter the processor names.

Also, the CAPS user interface should be modified to allow user to enter the value *max_load_factor* (refer to Building the Distributed Schedule in Chapter IV, Section C), which is used by the scheduler to set the maximum allowed load factor for each processor.

LIST OF REFERENCES

ACT Europe. undated. *Glade User Manual*, Paris and Brittany: Ada Core Technologies. http://www.gnat.com

Ada. 1995. The Language Reference Manual. Intermetrics, Inc.

Audsley, N. and A. Burns. 1983. *Real-Time System Scheduling*. Technical Paper, University of York, UK.

Awad, Maher, Juha Kuusela and Jurgen Ziegler. 1996. *Object-Oriented Technology For Real-Time Systems: A Practical Approach Using OMT And Fusion*, Upper Saddle River, NJ: Prentice-Hall, Inc.

Baker, K. 1974. *Introduction to Sequencing and Scheduling*, New York: John Wiley and Sons, Inc.

Bierrel, A. D. and B. J Nelson. 1984. "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems* 2: 39-59.

Blazewicz, J. 1976. "Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines." In *Proceedings of the International Workshop on Modeling and Performance Evaluation of Computer Systems*, Amsterdam: North-Holland.

Cheng, S. C., J. A. Stankovic, and K. Ramamritham. 1987. *Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey*, COINS Technical Report 87-55, June 10.

Cheng, S. T., and A. K. Agrawala. 1993. *Scheduling of Periodic Tasks with Relative Timing Constraints* (Technical Report CS-TR-3392), College Park, MD: Department of Computer Science, University of Maryland.

Cheng, S. T., and A. K. Agrawala. 1995. *Allocation and Scheduling of Real-Time Periodic Tasks with Relative Timing Constraints* (Technical Report CS-TR-3402), College Park, MD: Department of Computer Science, University of Maryland.

Chu, W. W., L. Y. Holloway, M. T. Lan and K. Efe, 1980. "Task Allocation in Distributed Data Processing," *Computer* 13 (11): 57-69.

Cordeiro, Mauricio de Menezes. 1995. "Distributed Hard Real-Time Scheduling for a Software Prototyping Environment." Ph.D. Dissertation, Department of Computer Science. Monterey, CA: United States Naval Postgraduate School.

Coulouris, G., Dollimore, J. and Kindberg, T. 1996. *Distributed Systems: Concepts and Design*, Reading, MA: Addison-Wesley.

Garey, M. R. and D. S. Johnson. 1977. "Two-processors Scheduling with Start-times and Deadlines," *SIAM Journal on Computing* 6: 416-426.

International Standards Organization. 1992. *Basic Reference Model of Open Distributed Processing, Part 1: Overview and Guide to Use* (ISO/IEC JTC1/SC212/WG7 CD10746-1), International Standards Organization.

Jeffay, K., D. Stanat, and C. Martel. 1991. "On Non-Preemptive Scheduling of Periodic and Sporadic Tasks." In *Proceedings of Real-Time Systems Symposium*, December.

Jenny, C. J. 1977. "Process Partioning in Distributed Systems." In *Digest of Papers National Telecommunications Conference*.

Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi. 1983. "Optimization by Simulated Annealing," *Science* 220 (May 1983): 671-680.

Lageweg, B. J, J. K. Lenstra and A.H.G. Rinnooy Kan. 1976. *Minimizing Maximum Lateness on One Machine: Computational Experience and Some Applications*, Statistica Neerlandica 30: 25-41.

Laplante, Phillip A. 1993. Real-Time Systems Design and Analysis: An Engineer's Handbook. New York: IEEE Press.

Lawler, E. L. 1973. "Optimal Sequencing of a Single Machine Subject to Precedence Constraints," *Management Science*, 19: 544-546.

Lin, F. T., and C. C. Hsu. 1991. "Task Assignment Problems in Distributed Computing Systems by Simulated Annealing," *Journal of the Chinese Institute of Engineers* 14(5): 537-550.

Liskov, B. 1988. "Distributed Programming in Argus," *Communications of the Association for Computing Machinery (ACM)* 31(3): 300-12.

Liskov, B., and Scheifler, R. W. 1982. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *Association for Computing Machinery (ACM) Transactions on Programming Languages and Systems* 5(3): 381-404.

Liu, C. L., and J. W. Layland. 1973. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the Association for Computing Machinery (ACM)* 20(1): 46-61.

Luqi. 1989. "Handling Timing Constraints in Rapid Prototyping." In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Science*, Kailua-Kona, HI: Computer Society Press, January.

Luqi. 1992. "Computer-Aided Prototyping for a Command-and-Control System Using CAPS," *IEEE Software*, January.

Luqi. 1993. "Real-Time Constraints in a Rapid Prototyping Language," *Computer Language* 18 (2): 77-103.

Luqi and M. Shing. 1996. "Real-Time Scheduling for Software Prototyping," *Journal of Systems Integration* 6: 41-72.

Luqi, M. Shing, and J. Brockett. 1993. "Real-Time Scheduling in System Prototyping." In *Proceedings of the Fourth International Workshop on Rapid System Prototyping*. Research Triangle Park, NC, June.

Luqi, V. Berzins and R. T. Yeh. 1988. "A Prototyping Language for Real-time Software," *IEEE Transactions on Software Engineering* 14(10): 1409-1423.

Mok, A. K. 1983. "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment." Ph.D. Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Ramamritham, Krithi. 1990. "Allocation and Scheduling of Complex Periodic Tasks." In *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, FR.

Shatz, S. M. 1984. "Communication Mechanisms for Programming Distributed Systems," *IEEE Computer,* June: 21-28.

Shatz, S. M. and J. Wang. 1988. *Tutorial: Distributed Software Engineering*. Washington, DC: IEEE Computer Society Press.

Sinha, Pradeep K. 1997. *Distributed Operating Systems*. New York: IEEE Press.

Stankovic J. A., and K. Ramamritham. 1988. *Tutorial on Hard Real-Time Systems*. Washington, DC: IEEE Computer Society Press.

Tanenbaum, A. S. 1995. *Distributed Operating Systems*. Englewood Cliffs, NJ: Prentice-Hall.

Tindell, K. W., A. Burns, and A. J. Wellings. 1992. "Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy," *Real-Time Systems* 4(2): 145-165.

Ullman, J. D. 1976. "Complexity of Sequence Problem." In *Computer and Job-Shop Scheduling Theory*, ed. E. G. Coffman, New York: John Wiley and Sons.

Zhu, J., T. G. Lewis and J. Colin. *Scheduling Hard Real-Time Constrained Tasks on One Processor*, To be published.

APPENDIX A. SPECIFICATION OF THE DISTRIBUTED

AUTOPILOT ATOMIC OPERATORS

```
-- Unit          : autopilot.altimeter
-- Prototype     : CAPS autopilot
-- Date          : June 94
-- Author        : Jim Brockett
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : altimeter Ada implementation
-- Notes         : Adapted to the distributed implementation by Jose Carlos Almeida
--                 September 1998
```

```
package altimeter_PKG is

  MAXIMUM_ALTITUDE : Integer := 35000;
  MINIMUM_ALTITUDE : Integer := 0;

  procedure altimeter(actual_altitude : out Integer;
                      delta_altitude  : in Integer);

end altimeter_PKG;
```

```
-- Unit          : autopilot.correct_altitude
-- Prototype     : CAPS autopilot
-- Date          : June `94
-- Author        : Jim Brockett
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : correct_altitude Ada implementation
-- Notes         : Adapted to the distributed implementation by Jose Carlos Almeida  September 1998
```

```
with altitude_command_type_PKG; use altitude_command_type_PKG;

package correct_altitude_PKG is

  procedure correct_altitude(actual_altitude  : in Integer;
                             desired_altitude : in Integer;
                             altitude_command : out altitude_command_type);

end correct_altitude_PKG;
```

```
-------------------------------------------------------------------------------
-- Unit          : autopilot.control_elevator
-- Prototype     : CAPS autopilot
-- Date          : June `94
-- Author        : Jim Brockett
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : control_elevator Ada implementation
-- Notes         : Adapted to the distributed implementation by Jose Carlos Almeida
--                  September 1998
--                  This unit was created for the distributed Autopilot prototype by spliting the original
--                  operator control_surfaces in the uniprocessor implementation
-------------------------------------------------------------------------------

with altitude_command_type_PKG; use altitude_command_type_PKG;
with elevator_status_type_PKG; use elevator_status_type_PKG;

package control_elevator_PKG is

  procedure control_elevator(altitude_command : in altitude_command_type;
                             elevator_status  : out elevator_status_type;
                             delta_altitude   : out INTEGER);

end control_elevator_PKG;


-------------------------------------------------------------------------------
-- Unit          : autopilot.display_1
-- Prototype     : CAPS autopilot
-- Date          : September 1998
-- Author        : Jose Carlos Almeida
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : Display and input data Ada implementation
-- Notes         : This unit was created for the distributed Autopilot implementation
-------------------------------------------------------------------------------

-- Add "with" and "use" statements for user-defined types used by operator display_1
with elevator_status_type_PKG; use elevator_status_type_PKG;
with rudder_status_type_PKG; use rudder_status_type_PKG;

package display_1_PKG is

  procedure display_1(actual_altitude  : in INTEGER;
                      elevator_status  : in elevator_status_type;
                      rudder_status    : in rudder_status_type;
                      desired_altitude : out INTEGER);

end display_1_PKG;
```

```
-----------------------------------------------------------------------------------------------------
-- Unit          : autopilot.compass
-- Prototype      : CAPS autopilot
-- Date          : June `94
-- Author         : Jim Brockett
-- Compiler       : GLADE 1.03p/Gnat 3.10p
-- Description     : compass Ada implementation
-- Notes          : Adapted to the distributed implementation by Jose Carlos Almeida
--                 September 1998
-----------------------------------------------------------------------------------------------------

package compass_PKG is

  procedure compass(delta_course  : in Integer;
                    actual_course : out Integer);

end compass_PKG;


-----------------------------------------------------------------------------------------------------
-- Unit          : autopilot.correct_course
-- Prototype      : CAPS autopilot
-- Date          : June `94
-- Author         : Jim Brockett
-- Compiler       : GLADE 1.03p/Gnat 3.10p
-- Description     : correct_course Ada implementation
-- Notes          : Adapted to the distributed implementation by Jose Carlos Almeida
--                 September 1998
-----------------------------------------------------------------------------------------------------

with course_command_type_PKG; use course_command_type_PKG;

package correct_course_PKG is

  procedure correct_course(desired_course : in Integer;
                           actual_course  : in Integer;
                           course_command : out course_command_type);

end correct_course_PKG;
```

```
---------------------------------------------------------------------------------
-- Unit           :  autopilot.control_rudder
-- Prototype      :  CAPS autopilot
-- Date           :  June `94
-- Author         :  Jim Brockett
-- Compiler       :  GLADE 1.03p/Gnat 3.10p
-- Description     :  control_rudder Ada implementation
-- Notes          :  Adapted to the distributed implementation by Jose Carlos Almeida
--                   September 1998
--                   This unit was created for the distributed Autopilot prototype by spliting the original
--                   operator control_surfaces in the uniprocessor implementation
---------------------------------------------------------------------------------

with course_command_type_PKG; use course_command_type_PKG;
with rudder_status_type_PKG; use rudder_status_type_PKG;

package control_rudder_PKG is

  procedure control_rudder(course_command  : in course_command_type;
                           rudder_status    : out rudder_status_type;
                           delta_course     : out INTEGER);


end control_rudder_PKG;


---------------------------------------------------------------------------------
-- Unit           :  autopilot.display_2
-- Prototype      :  CAPS autopilot
--- Date          :  September 1998
-- Author         :  Jose Carlos Almeida
-- Compiler       :  GLADE 1.03p/Gnat 3.10p
-- Description     :  display and input data Ada implementation
-- Notes          :  This unit was created for the distributed Autopilot implementation
---------------------------------------------------------------------------------

-- Add "with" and "use" statements for user-defined types used by operator display_2
with elevator_status_type_PKG; use elevator_status_type_PKG;
with rudder_status_type_PKG; use rudder_status_type_PKG;

package display_2_PKG is

  procedure display_2(actual_course   : in INTEGER;
                      desired_course  : out INTEGER);

end display_2_PKG;
```

APPENDIX B. DISTRIBUTED AUTOPILOT STREAMS INSTANTIATIONS

```
---------------------------------------------------------------------------------------
-- Unit          : autopilot_1_streams
-- Prototype     : CAPS autopilot
-- Date          : June 94
-- Author        : Jim Brockett
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : Distributed Autopilot streams instantiations
-- Notes         : Adapted to the distributed implementation by Jose Carlos Almeida
--                   September 1998
---------------------------------------------------------------------------------------


--Streams declarations for Partition 1

-- with/use clauses for atomic type packages
with ELEVATOR_STATUS_TYPE_PKG; use ELEVATOR_STATUS_TYPE_PKG;
with RUDDER_STATUS_TYPE_PKG; use RUDDER_STATUS_TYPE_PKG;
with ALTITUDE_COMMAND_TYPE_PKG; use ALTITUDE_COMMAND_TYPE_PKG;

-- with/use clauses for generated packages.
with AUTOPILOT_EXCEPTIONS; use AUTOPILOT_EXCEPTIONS;
with AUTOPILOT_1_INSTANTIATIONS; use AUTOPILOT_1_INSTANTIATIONS;

-- with/use clauses for CAPS library packages.
with PSDL_STREAMS; use PSDL_STREAMS;

package AUTOPILOT_1_STREAMS is

-- Local stream instantiations

package DS_ACTUAL_ALTITUDE_DISPLAY_1 is new
    PSDL_STREAMS.SAMPLED_BUFFER(INTEGER);

package DS_ACTUAL_ALTITUDE_CORRECT_ALTITUDE is new
    PSDL_STREAMS.FIFO_BUFFER(INTEGER);

package DS_ALTITUDE_COMMAND_CONTROL_ELEVATOR is new
    PSDL_STREAMS.SAMPLED_BUFFER(ALTITUDE_COMMAND_TYPE);

package DS_ELEVATOR_STATUS_DISPLAY_1 is new
    PSDL_STREAMS.SAMPLED_BUFFER(ELEVATOR_STATUS_TYPE);

-- State stream instantiations package

package DS_DESIRED_ALTITUDE_CORRECT_ALTITUDE is new
    PSDL_STREAMS.STATE_VARIABLE(INTEGER, 0);

package DS_DELTA_ALTITUDE_ALTIMETER is new
    PSDL_STREAMS.STATE_VARIABLE(INTEGER, 0);

package DS_RUDDER_STATUS_DISPLAY_1 is new
    PSDL_STREAMS.STATE_VARIABLE(RUDDER_STATUS_TYPE, straight);

end AUTOPILOT_1_STREAMS;
```

129

```
--------------------------------------------------------------------------------
-- Unit          : autopilot_2_streams
-- Prototype      : CAPS autopilot
-- Date           : June 94
-- Author         : Jim Brockett
-- Compiler       : GLADE 1.03p/Gnat 3.10p
-- Description     : Distributed Autopilot streams instantiations
-- Notes           : Adapted to the distributed implementation by Jose Carlos Almeida
--                     September 1998
--------------------------------------------------------------------------------

--Streams declarations for Partition 2

-- with/use clauses for atomic type packages
with COURSE_COMMAND_TYPE_PKG; use COURSE_COMMAND_TYPE_PKG;
with ALTITUDE_COMMAND_TYPE_PKG; use ALTITUDE_COMMAND_TYPE_PKG;

-- with/use clauses for generated packages.
with AUTOPILOT_EXCEPTIONS; use AUTOPILOT_EXCEPTIONS;
with AUTOPILOT_2_INSTANTIATIONS; use AUTOPILOT_2_INSTANTIATIONS;

-- with/use clauses for CAPS library packages.
with PSDL_STREAMS; use PSDL_STREAMS;

package AUTOPILOT_2_STREAMS is

  -- Local stream instantiations

  package DS_ACTUAL_COURSE_CORRECT_COURSE is new
      PSDL_STREAMS.FIFO_BUFFER(INTEGER);

  package DS_ACTUAL_COURSE_DISPLAY_2 is new
      PSDL_STREAMS.FIFO_BUFFER(INTEGER);

  package DS_COURSE_COMMAND_CONTROL_RUDDER is new
      PSDL_STREAMS.SAMPLED_BUFFER(COURSE_COMMAND_TYPE);

  -- State stream instantiations

  package DS_DELTA_COURSE_COMPASS is new
      PSDL_STREAMS.STATE_VARIABLE(INTEGER, 0);

  package DS_DESIRED_COURSE_CORRECT_COURSE is new
      PSDL_STREAMS.STATE_VARIABLE(INTEGER, 0);

end AUTOPILOT_2_STREAMS;
```

130

APPENDIX C. DISTRIBUTED AUTOPILOT REMOTE STREAMS

```
-------------------------------------------------------------------------------------------
-- Unit          : autopilot_1_remote_streams
-- Prototype     : CAPS autopilot
-- Date          : September 1998
-- Author        : Jose Carlos Almeida
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : Remote Call Interface for partition 1 in the distributed autopilot prototype
-- Notes         : This package is called whenever remote operators in partititon 2 need to communicate
--                       with operators in partition 1
-------------------------------------------------------------------------------------------
with rudder_status_type_pkg; use rudder_status_type_pkg;

package AutoPilot_1_REMOTE_STREAMS is

 pragma Remote_Call_Interface;


 procedure Write_Rudder_Status_Display_1(Rudder_Status: in rudder_status_type);

 -- Allow interprocessor comunication using asynchronous RPCs
 pragma Asynchronous(Write_Rudder_Status_Display_1);

end AutoPilot_1_REMOTE_STREAMS;
```

```
-------------------------------------------------------------------------------------------
-- Unit          : autopilot_1_remote_streams
-- Prototype     : CAPS autopilot
-- Date          : September 1998
-- Author        : Jose Carlos Almeida
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : Remote Call Interface for partition 1 in the distributed autopilot prototype
-- Notes         : This package is called whenever remote operators in partititon 2 need to communicate
--                       with operators in partition 1
-------------------------------------------------------------------------------------------
with AUTOPILOT_1_STREAMS; use AUTOPILOT_1_STREAMS;
with Autopilot_1_Start_Drivers; use Autopilot_1_Start_Drivers;

package body AutoPilot_1_REMOTE_STREAMS is

 procedure Write_Rudder_Status_Display_1(Rudder_Status: in rudder_status_type) is

  begin

  -- Write Rudder_Status to the corresponding producer operator output stream
    WRITE_RUDDER_STATUS(Rudder_Status);

 end Write_Rudder_Status_Display_1;

end AutoPilot_1_REMOTE_STREAMS;
```

APPENDIX D. DISTRIBUTED AUTOPILOT DRIVERS

```
-------------------------------------------------------------------------------
-- Unit          : autopilot_1_drivers
-- Prototype     : CAPS autopilot
-- Date          : June 94
-- Author        : Jim Brockett
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : This package reads the input streams, invoke the corresponding operators and writes
--                  data to the corresponding output stream
-- Notes          : Adapted to the distributed implementation by Jose Carlos Almeida September 1998
-------------------------------------------------------------------------------

package AUTOPILOT_1_DRIVERS is
  procedure ALTIMETER_DRIVER;
  procedure CORRECT_ALTITUDE_DRIVER;
  procedure CONTROL_ELEVATOR_DRIVER;
  procedure DISPLAY_1_DRIVER;
end AUTOPILOT_1_DRIVERS;


-------------------------------------------------------------------------------
-- Unit          : autopilot_1_drivers
-- Prototype     : CAPS autopilot
-- Date          : June 94
-- Author        : Jim Brockett
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : This package reads the input streams, invoke the corresponding operators and writes
--                  data to the corresponding output stream in partition 1
-- Notes          : Adapted to the distributed implementation by Jose Carlos Almeida September 1998
-------------------------------------------------------------------------------

-- with/use clauses for atomic components.
with RUDDER_STATUS_TYPE_PKG; use RUDDER_STATUS_TYPE_PKG;
with ALTITUDE_COMMAND_TYPE_PKG; use ALTITUDE_COMMAND_TYPE_PKG;
with ELEVATOR_STATUS_TYPE_PKG; use ELEVATOR_STATUS_TYPE_PKG;
with ALTIMETER_PKG; use ALTIMETER_PKG;
with CORRECT_ALTITUDE_PKG; use CORRECT_ALTITUDE_PKG;
with CONTROL_ELEVATOR_PKG; use CONTROL_ELEVATOR_PKG;
with DISPLAY_1_PKG; use DISPLAY_1_PKG;


-- with/use clauses for generated packages.
with AUTOPILOT_EXCEPTIONS; use AUTOPILOT_EXCEPTIONS;
with AUTOPILOT_1_STREAMS; use AUTOPILOT_1_STREAMS;
with AUTOPILOT_1_TIMERS; use AUTOPILOT_1_TIMERS;
with AUTOPILOT_1_INSTANTIATIONS; use AUTOPILOT_1_INSTANTIATIONS;

-- with/use clauses for CAPS library packages.
with DS_DEBUG_PKG; use DS_DEBUG_PKG;
with PSDL_STREAMS; use PSDL_STREAMS;
with PSDL_TIMERS;

-- with/use clauses for RCI packages
```

```
package body AUTOPILOT_1_DRIVERS is

procedure ALTIMETER_DRIVER is
 LV_DELTA_ALTITUDE : INTEGER;
 LV_ACTUAL_ALTITUDE : INTEGER;

 EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
 EXCEPTION_ID: PSDL_EXCEPTION;
 begin
  -- Data trigger checks.

  -- Data stream reads.
  begin
   DS_DELTA_ALTITUDE_ALTIMETER.BUFFER.READ(LV_DELTA_ALTITUDE);
  exception
   when BUFFER_UNDERFLOW =>
    DS_DEBUG.BUFFER_UNDERFLOW("DELTA_ALTITUDE_ALTIMETER", "ALTIMETER");
  end;

  -- Execution trigger condition check.
  if True then
   begin
    ALTIMETER(
    DELTA_ALTITUDE => LV_DELTA_ALTITUDE,
    ACTUAL_ALTITUDE => LV_ACTUAL_ALTITUDE);
    exception
    when others =>
     DS_DEBUG.UNDECLARED_EXCEPTION("ALTIMETER");
     EXCEPTION_HAS_OCCURRED := true;
     EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
   end;
  else return;
  end if;

  -- Exception Constraint translations.

  -- Other constraint option translations.

  --Unconditional output translations.
  if not EXCEPTION_HAS_OCCURRED then
   begin

    DS_ACTUAL_ALTITUDE_DISPLAY_1.BUFFER.WRITE(LV_ACTUAL_ALTITUDE);
    exception
    when BUFFER_OVERFLOW =>
     DS_DEBUG.BUFFER_OVERFLOW("ACTUAL_ALTITUDE_DISPLAY", "ALTIMETER");
   end;
   begin
    DS_ACTUAL_ALTITUDE_CORRECT_ALTITUDE.BUFFER.WRITE
                                        (LV_ACTUAL_ALTITUDE);
    exception
    when BUFFER_OVERFLOW =>
      DS_DEBUG.BUFFER_OVERFLOW("ACTUAL_ALTITUDE_CORRECT_ALTITUDE",
                                        "ALTIMETER");
   end;
  end if;
```

134

```
     -- PSDL Exception handler.
     if EXCEPTION_HAS_OCCURRED then
      DS_DEBUG.UNHANDLED_EXCEPTION(
       "ALTIMETER",
       PSDL_EXCEPTION'IMAGE(EXCEPTION_ID));
     end if;
end ALTIMETER_DRIVER;


-------------------------------------------------------------------------------------------------
--procedure CORRECT_ALTITUDE_DRIVER
-------------------------------------------------------------------------------------------------

procedure CORRECT_ALTITUDE_DRIVER is
 LV_ACTUAL_ALTITUDE : INTEGER;
 LV_DESIRED_ALTITUDE : INTEGER;
 LV_ALTITUDE_COMMAND :
            ALTITUDE_COMMAND_TYPE_PKG.ALTITUDE_COMMAND_TYPE;

 EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
 EXCEPTION_ID: PSDL_EXCEPTION;
 begin
  -- Data trigger checks.
  if not (DS_ACTUAL_ALTITUDE_CORRECT_ALTITUDE.BUFFER.NEW_DATA) then
   return;
  end if;

  -- Data stream reads.
  begin
   DS_ACTUAL_ALTITUDE_CORRECT_ALTITUDE.BUFFER.READ(LV_ACTUAL_ALTITUDE);
   exception
   when BUFFER_UNDERFLOW =>
    DS_DEBUG.BUFFER_UNDERFLOW("ACTUAL_ALTITUDE_CORRECT_ALTITUDE",
                                              "CORRECT_ALTITUDE");
  end;
  begin
   DS_DESIRED_ALTITUDE_CORRECT_ALTITUDE.BUFFER.READ
                                              (LV_DESIRED_ALTITUDE);

   exception
   when BUFFER_UNDERFLOW =>
    DS_DEBUG.BUFFER_UNDERFLOW("DESIRED_ALTITUDE_CORRECT_ALTITUDE",
                                              "CORRECT_ALTITUDE");
  end;

  -- Execution trigger condition check.
  if True then
   begin
    CORRECT_ALTITUDE(
    ACTUAL_ALTITUDE => LV_ACTUAL_ALTITUDE,
    DESIRED_ALTITUDE => LV_DESIRED_ALTITUDE,
    ALTITUDE_COMMAND => LV_ALTITUDE_COMMAND);
    exception
    when others =>
     DS_DEBUG.UNDECLARED_EXCEPTION("CORRECT_ALTITUDE");
     EXCEPTION_HAS_OCCURRED := true;
     EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
   end;
  else return;
```

135

```
     end if;

     -- Exception Constraint translations.

     -- Other constraint option translations.

     --Unconditional output translations.
     if not EXCEPTION_HAS_OCCURRED then
       begin
         DS_ALTITUDE_COMMAND_CONTROL_ELEVATOR.BUFFER.WRITE
                                                 (LV_ALTITUDE_COMMAND);

       exception
       when BUFFER_OVERFLOW =>
         DS_DEBUG.BUFFER_OVERFLOW("ALTITUDE_COMMAND_CONTROL_ELEVATOR",
                                             "CORRECT_ALTITUDE");

       end;
     end if;

     -- PSDL Exception handler.
     if EXCEPTION_HAS_OCCURRED then
       DS_DEBUG.UNHANDLED_EXCEPTION(
       "CORRECT_ALTITUDE",
       PSDL_EXCEPTION'IMAGE(EXCEPTION_ID));
     end if;
end CORRECT_ALTITUDE_DRIVER;

---------------------------------------------------------------------------------
-- procedure CONTROL_ELEVATOR_DRIVER
---------------------------------------------------------------------------------

procedure CONTROL_ELEVATOR_DRIVER is
 LV_ALTITUDE_COMMAND :
       ALTITUDE_COMMAND_TYPE_PKG.ALTITUDE_COMMAND_TYPE;
 LV_ELEVATOR_STATUS : ELEVATOR_STATUS_TYPE_PKG.ELEVATOR_STATUS_TYPE;
 LV_DELTA_ALTITUDE : INTEGER;

EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
EXCEPTION_ID: PSDL_EXCEPTION;
begin
  -- Data trigger checks.

  -- Data stream reads.

  begin
   DS_ALTITUDE_COMMAND_CONTROL_ELEVATOR.BUFFER.READ
                                             (LV_ALTITUDE_COMMAND);
  exception
  when BUFFER_UNDERFLOW =>
     DS_DEBUG.BUFFER_UNDERFLOW("ALTITUDE_COMMAND_CONTROL_ELEVATOR",
                                             "CONTROL_ELEVATOR");
  end;

  -- Execution trigger condition check.
  if True then
   begin
```

```
        CONTROL_ELEVATOR(
        ALTITUDE_COMMAND => LV_ALTITUDE_COMMAND,
        ELEVATOR_STATUS => LV_ELEVATOR_STATUS,
        DELTA_ALTITUDE => LV_DELTA_ALTITUDE);
        exception
        when others =>
          DS_DEBUG.UNDECLARED_EXCEPTION("CONTROL_ELEVATOR");
          EXCEPTION_HAS_OCCURRED := true;
          EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
      end;
    else return;
    end if;

    -- Exception Constraint translations.

    -- Other constraint option translations.

    -- Unconditional output translations.

    if not EXCEPTION_HAS_OCCURRED then
      begin
        DS_ELEVATOR_STATUS_DISPLAY_1.BUFFER.WRITE(LV_ELEVATOR_STATUS);
        exception
        when BUFFER_OVERFLOW =>
          DS_DEBUG.BUFFER_OVERFLOW("ELEVATOR_STATUS_DISPLAY",
                                                "CONTROL_SURFACES");
      end;
    end if;

    if not EXCEPTION_HAS_OCCURRED then
      begin
        DS_DELTA_ALTITUDE_ALTIMETER.BUFFER.WRITE(LV_DELTA_ALTITUDE);
        exception
        when BUFFER_OVERFLOW =>
          DS_DEBUG.BUFFER_OVERFLOW("DELTA_ALTITUDE_ALTIMETER",
                                                "CONTROL_SURFACES");
      end;
    end if;

    -- PSDL Exception handler.
    if EXCEPTION_HAS_OCCURRED then
      DS_DEBUG.UNHANDLED_EXCEPTION(
        "CONTROL_ELEVATOR", SDL_EXCEPTION'IMAGE(EXCEPTION_ID));
    end if;
end CONTROL_ELEVATOR_DRIVER;


-----------------------------------------------------------------------------
-- procedure DISPLAY_1_DRIVER
-----------------------------------------------------------------------------
procedure DISPLAY_1_DRIVER is
  LV_ACTUAL_ALTITUDE : INTEGER;
  LV_ELEVATOR_STATUS : ELEVATOR_STATUS_TYPE_PKG.ELEVATOR_STATUS_TYPE;
  LV_RUDDER_STATUS : RUDDER_STATUS_TYPE_PKG.RUDDER_STATUS_TYPE;
  LV_DESIRED_ALTITUDE : INTEGER;

  EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
```

```
EXCEPTION_ID: PSDL_EXCEPTION;
begin
 -- Data trigger checks.

 -- Data stream reads.
 begin
  DS_ACTUAL_ALTITUDE_DISPLAY_1.BUFFER.READ(LV_ACTUAL_ALTITUDE);
  exception
  when BUFFER_UNDERFLOW =>
   DS_DEBUG.BUFFER_UNDERFLOW("ACTUAL_ALTITUDE_DISPLAY_1", "DISPLAY_1");
 end;
 begin
  DS_ELEVATOR_STATUS_DISPLAY_1.BUFFER.READ(LV_ELEVATOR_STATUS);
 exception
  when BUFFER_UNDERFLOW =>
   DS_DEBUG.BUFFER_UNDERFLOW("ELEVATOR_STATUS_DISPLAY_1", "DISPLAY_1");
 end;
 begin
  DS_RUDDER_STATUS_DISPLAY_1.BUFFER.READ(LV_RUDDER_STATUS);
 exception
  when BUFFER_UNDERFLOW =>
   DS_DEBUG.BUFFER_UNDERFLOW("RUDDER_STATUS_DISPLAY_1", "DISPLAY_1");
 end;

 -- Execution trigger condition check.
 if True then
  begin
  DISPLAY_1(
   ACTUAL_ALTITUDE => LV_ACTUAL_ALTITUDE,
   ELEVATOR_STATUS => LV_ELEVATOR_STATUS,
   RUDDER_STATUS => LV_RUDDER_STATUS,
   DESIRED_ALTITUDE => LV_DESIRED_ALTITUDE);

   exception
   when others =>
    DS_DEBUG.UNDECLARED_EXCEPTION("DISPLAY_1");
    EXCEPTION_HAS_OCCURRED := true;
    EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
  end;
 else return;
 end if;

 -- Exception Constraint translations.

 -- Other constraint option translations.

 --Unconditional output translations.
 if not EXCEPTION_HAS_OCCURRED then
  begin
   DS_DESIRED_ALTITUDE_CORRECT_ALTITUDE.BUFFER.WRITE
                                                (LV_DESIRED_ALTITUDE);
  exception
   when BUFFER_OVERFLOW =>
    DS_DEBUG.BUFFER_OVERFLOW("DESIRED_ALTITUDE_CORRECT_ALTITUDE",
                                              "DISPLAY_1");
  end;
```

138

```
      end if;

      -- PSDL Exception handler.
      if EXCEPTION_HAS_OCCURRED then
        DS_DEBUG.UNHANDLED_EXCEPTION(
        "DISPLAY_1", PSDL_EXCEPTION'IMAGE(EXCEPTION_ID));
      end if;
    end DISPLAY_1_DRIVER;

end AUTOPILOT_1_DRIVERS;
```

```
-----------------------------------------------------------------------------------------------------------
-- Unit          : autopilot_2_drivers
-- Prototype      : CAPS autopilot
-- Date           : June 94
-- Author         : Jim Brockett
-- Compiler       : GLADE 1.03p/Gnat 3.10p
-- Description     : This package reads the input streams, invoke the corresponding operators and writes
--                   data to the corresponding output stream
-- Notes          : Adapted to the distributed implementation by Jose Carlos Almeida September 1998
-----------------------------------------------------------------------------------------------------------
```

```
package AUTOPILOT_2_DRIVERS is

  procedure COMPASS_DRIVER;
  procedure CORRECT_COURSE_DRIVER;
  procedure CONTROL_RUDDER_DRIVER;
  procedure DISPLAY_2_DRIVER;

end AUTOPILOT_2_DRIVERS;
```

```
-----------------------------------------------------------------------------------------------------------
-- Unit          : autopilot_2_drivers
-- Prototype      : CAPS autopilot
-- Date           : June 94
-- Author         : Jim Brockett
-- Compiler       : GLADE 1.03p/Gnat 3.10p
-- Description     : This package reads the input streams, invokes the corresponding operators and writes
--                   data to the corresponding output stream
-- Notes          : Adapted to the distributed implementation by Jose Carlos Almeida September 1998
-----------------------------------------------------------------------------------------------------------
-- with/use clauses for atomic components.
with COURSE_COMMAND_TYPE_PKG; use COURSE_COMMAND_TYPE_PKG;
with RUDDER_STATUS_TYPE_PKG; use RUDDER_STATUS_TYPE_PKG;
with CORRECT_COURSE_PKG; use CORRECT_COURSE_PKG;
with CONTROL_RUDDER_PKG; use CONTROL_RUDDER_PKG;
with COMPASS_PKG; use COMPASS_PKG;
with DISPLAY_2_PKG; use DISPLAY_2_PKG;

-- with/use clauses for generated packages.
with AUTOPILOT_EXCEPTIONS; use AUTOPILOT_EXCEPTIONS;
with AUTOPILOT_2_STREAMS; use AUTOPILOT_2_STREAMS;
with AUTOPILOT_2_TIMERS; use AUTOPILOT_2_TIMERS;
with AUTOPILOT_2_INSTANTIATIONS; use AUTOPILOT_2_INSTANTIATIONS;

-- with/use clauses for CAPS library packages.
```

139

```ada
with DS_DEBUG_PKG; use DS_DEBUG_PKG;
with PSDL_STREAMS; use PSDL_STREAMS;
with PSDL_TIMERS;

-- with/use clauses for RCI packages
with AutoPilot_1_REMOTE_STREAMS; use AutoPilot_1_REMOTE_STREAMS;

package body AUTOPILOT_2_DRIVERS is

procedure COMPASS_DRIVER is
  LV_DELTA_COURSE : INTEGER;
  LV_ACTUAL_COURSE : INTEGER;

  EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
  EXCEPTION_ID: PSDL_EXCEPTION;
begin
  -- Data trigger checks.

  -- Data stream reads.
  begin
    DS_DELTA_COURSE_COMPASS.BUFFER.READ(LV_DELTA_COURSE);
  exception
    when BUFFER_UNDERFLOW =>
      DS_DEBUG.BUFFER_UNDERFLOW("DELTA_COURSE_COMPASS", "COMPASS");
  end;

  -- Execution trigger condition check.
  if True then
    begin
      COMPASS(
        DELTA_COURSE => LV_DELTA_COURSE,
        ACTUAL_COURSE => LV_ACTUAL_COURSE);
    exception
    when others =>
      DS_DEBUG.UNDECLARED_EXCEPTION("COMPASS");
      EXCEPTION_HAS_OCCURRED := true;
      EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
    end;
  else return;
  end if;

  -- Exception Constraint translations.

  -- Other constraint option translations.

  -- Unconditional output translations.
  if not EXCEPTION_HAS_OCCURRED then
    begin
      DS_ACTUAL_COURSE_CORRECT_COURSE.BUFFER.WRITE(LV_ACTUAL_COURSE);
    exception
    when BUFFER_OVERFLOW =>
      DS_DEBUG.BUFFER_OVERFLOW("ACTUAL_COURSE_CORRECT_COURSE",
                                                            "COMPASS");
    end;
    begin
      DS_ACTUAL_COURSE_DISPLAY_2.BUFFER.WRITE(LV_ACTUAL_COURSE);
```

140

```
      exception
      when BUFFER_OVERFLOW =>
       DS_DEBUG.BUFFER_OVERFLOW("ACTUAL_COURSE_DISPLAY_2", "COMPASS");
      end;
     end if;

     -- PSDL Exception handler.
     if EXCEPTION_HAS_OCCURRED then
      DS_DEBUG.UNHANDLED_EXCEPTION(
       "COMPASS",
        PSDL_EXCEPTION'IMAGE(EXCEPTION_ID));
     end if;
end COMPASS_DRIVER;


----------------------------------------------------------------------------------------------------
-- procedure CORRECT_COURSE_DRIVER
----------------------------------------------------------------------------------------------------
procedure CORRECT_COURSE_DRIVER is
 LV_DESIRED_COURSE : INTEGER;
 LV_ACTUAL_COURSE : INTEGER;
 LV_COURSE_COMMAND : COURSE_COMMAND_TYPE_PKG.COURSE_COMMAND_TYPE;

 EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
 EXCEPTION_ID: PSDL_EXCEPTION;
begin
 -- Data trigger checks.
 if not (DS_ACTUAL_COURSE_CORRECT_COURSE.BUFFER.NEW_DATA) then
  return;
 end if;

 -- Data stream reads.
 begin
  DS_DESIRED_COURSE_CORRECT_COURSE.BUFFER.READ(LV_DESIRED_COURSE);
  exception
  when BUFFER_UNDERFLOW =>
   DS_DEBUG.BUFFER_UNDERFLOW("DESIRED_COURSE_CORRECT_COURSE",
                                          "CORRECT_COURSE");
 end;
 begin
  DS_ACTUAL_COURSE_CORRECT_COURSE.BUFFER.READ(LV_ACTUAL_COURSE);
  exception
  when BUFFER_UNDERFLOW =>
   DS_DEBUG.BUFFER_UNDERFLOW("ACTUAL_COURSE_CORRECT_COURSE",
                                          "CORRECT_COURSE");
 end;

 -- Execution trigger condition check.
 if True then
  begin
   CORRECT_COURSE(
     DESIRED_COURSE => LV_DESIRED_COURSE,
     ACTUAL_COURSE => LV_ACTUAL_COURSE,
     COURSE_COMMAND => LV_COURSE_COMMAND);
   exception
   when others =>
    DS_DEBUG.UNDECLARED_EXCEPTION("CORRECT_COURSE");
```

141

```
      EXCEPTION_HAS_OCCURRED := true;
      EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
   end;
 else return;
 end if;


 -- Exception Constraint translations.

 -- Other constraint option translations.

 -- Unconditional output translations.
 if not EXCEPTION_HAS_OCCURRED then
   begin
      DS_COURSE_COMMAND_CONTROL_RUDDER.BUFFER.WRITE
                                                   (LV_COURSE_COMMAND);
      exception
      when BUFFER_OVERFLOW =>
       DS_DEBUG.BUFFER_OVERFLOW("COURSE_COMMAND_CONTROL_RUDDER",
                                                   "CORRECT_COURSE");
   end;
   end if;


 -- PSDL Exception handler.
 if EXCEPTION_HAS_OCCURRED then
   DS_DEBUG.UNHANDLED_EXCEPTION(
     "CORRECT_COURSE",
     PSDL_EXCEPTION'IMAGE(EXCEPTION_ID));
   end if;
 end CORRECT_COURSE_DRIVER;


--------------------------------------------------------------------------------
-- procedure CONTROL_RUDDER_DRIVER
--------------------------------------------------------------------------------
procedure CONTROL_RUDDER_DRIVER is
 LV_COURSE_COMMAND : COURSE_COMMAND_TYPE_PKG.COURSE_COMMAND_TYPE;
 LV_RUDDER_STATUS : RUDDER_STATUS_TYPE_PKG.RUDDER_STATUS_TYPE;
 LV_DELTA_COURSE : INTEGER;

 EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
 EXCEPTION_ID: PSDL_EXCEPTION;
begin
 -- Data trigger checks.

 -- Data stream reads.
 begin
   DS_COURSE_COMMAND_CONTROL_RUDDER.BUFFER.READ(LV_COURSE_COMMAND);
   exception
   when BUFFER_UNDERFLOW =>
    DS_DEBUG.BUFFER_UNDERFLOW("COURSE_COMMAND_CONTROL_SURFACES",
                                                   "CONTROL_SURFACES");
 end;

 -- Execution trigger condition check.
 if True then
   begin
```

```
      CONTROL_RUDDER(
         COURSE_COMMAND => LV_COURSE_COMMAND,
         RUDDER_STATUS => LV_RUDDER_STATUS,
         DELTA_COURSE => LV_DELTA_COURSE);

   exception
   when others =>
      DS_DEBUG.UNDECLARED_EXCEPTION("CONTROL_RUDDER");
      EXCEPTION_HAS_OCCURRED := true;
      EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
  end;
else return;
end if;

-- Exception Constraint translations.

-- Other constraint option translations.

-- Unconditional output translations.
if not EXCEPTION_HAS_OCCURRED then
  begin
    -- need interprocessor communication
    -- write LV_RUDDER_STATUS to Autopilot_1_Remmote_Streams
    Write_Rudder_Status_Display_1(LV_RUDDER_STATUS);
  end;
end if;

if not EXCEPTION_HAS_OCCURRED then
  begin
    DS_DELTA_COURSE_COMPASS.BUFFER.WRITE(LV_DELTA_COURSE);
    exception
    when BUFFER_OVERFLOW =>
      DS_DEBUG.BUFFER_OVERFLOW("DELTA_COURSE_COMPASS",
                                                      "CONTROL_RUDDER");

  end;
end if;

-- PSDL Exception handler.
if EXCEPTION_HAS_OCCURRED then
  DS_DEBUG.UNHANDLED_EXCEPTION(
    "CONTROL_RUDDER",
    PSDL_EXCEPTION'IMAGE(EXCEPTION_ID));
end if;
end CONTROL_RUDDER_DRIVER;

-------------------------------------------------------------------------------
-- procedure DISPLAY_2_DRIVER
-------------------------------------------------------------------------------

procedure DISPLAY_2_DRIVER is
 LV_ACTUAL_COURSE   : INTEGER;
 LV_DESIRED_COURSE  : INTEGER;

 EXCEPTION_HAS_OCCURRED: BOOLEAN := FALSE;
 EXCEPTION_ID: PSDL_EXCEPTION;
begin
  -- Data trigger checks.
```

143

```
-- Data stream reads.
begin
 DS_ACTUAL_COURSE_DISPLAY_2.BUFFER.READ(LV_ACTUAL_COURSE);
 exception
 when BUFFER_UNDERFLOW =>
  DS_DEBUG.BUFFER_UNDERFLOW("ACTUAL_COURSE_DISPLAY_2", "DISPLAY_2");
end;

-- Execution trigger condition check.
if True then
 begin
  DISPLAY_2(
   ACTUAL_COURSE => LV_ACTUAL_COURSE,
   DESIRED_COURSE => LV_DESIRED_COURSE);
  exception
  when others =>
   DS_DEBUG.UNDECLARED_EXCEPTION("DISPLAY_2");
   EXCEPTION_HAS_OCCURRED := true;
   EXCEPTION_ID := UNDECLARED_ADA_EXCEPTION;
 end;
else return;
end if;

-- Exception Constraint translations.

-- Other constraint option translations.

--Unconditional output translations.
if not EXCEPTION_HAS_OCCURRED then
 begin
  DS_DESIRED_COURSE_CORRECT_COURSE.BUFFER.WRITE(LV_DESIRED_COURSE);
  exception
  when BUFFER_OVERFLOW =>
   DS_DEBUG.BUFFER_OVERFLOW("DESIRED_COURSE_CORRECT_COURSE",
                                              "DISPLAY_2");
 end;
end if;

-- PSDL Exception handler.
if EXCEPTION_HAS_OCCURRED then
 DS_DEBUG.UNHANDLED_EXCEPTION(
  "DISPLAY_2",
  PSDL_EXCEPTION'IMAGE(EXCEPTION_ID));
end if;
end DISPLAY_2_DRIVER;

end AUTOPILOT_2_DRIVERS;
```

APPENDIX E. DISTRIBUTED AUTOPILOT STATIC SCHEDULERS

```
---------------------------------------------------------------------------------------------------------------
-- Unit              : autopilot_1_static_schedulers
-- Prototype         : CAPS autopilot
-- Date              : June 94
-- Author            : Jim Brockett
-- Compiler          : GLADE 1.03p/Gnat 3.10p
-- Description        : This package invokes the drivers of the time critical operators in partition 1
-- Notes             : Adapted to the distributed implementation by Jose Carlos Almeida September 1998
---------------------------------------------------------------------------------------------------------------

package autopilot_1_STATIC_SCHEDULERS is
   procedure START_STATIC_SCHEDULE;
end autopilot_1_STATIC_SCHEDULERS;


---------------------------------------------------------------------------------------------------------------
-- Unit              : autopilot_1_static_schedulers
-- Prototype         : CAPS autopilot
-- Date              : June 94
-- Author            : Jim Brockett
-- Compiler          : GLADE 1.03p/Gnat 3.10p
-- Description        : This package invokes the drivers of the time critical operators in partition 1
-- Notes             : Adapted to the distributed implementation by Jose Carlos Almeida September 1998
---------------------------------------------------------------------------------------------------------------

with autopilot_1_DRIVERS;  use autopilot_1_DRIVERS;
with PRIORITY_DEFINITIONS;  use PRIORITY_DEFINITIONS;
with PSDL_TIMERS;  use PSDL_TIMERS;
with TEXT_IO;  use TEXT_IO;

package body autopilot_1_STATIC_SCHEDULERS is

 task type STATIC_SCHEDULE_TYPE is
  pragma priority (STATIC_SCHEDULE_PRIORITY);
  entry START;
 end STATIC_SCHEDULE_TYPE;
 for STATIC_SCHEDULE_TYPE'STORAGE_SIZE use 200_000;
 STATIC_SCHEDULE : STATIC_SCHEDULE_TYPE;

 task body STATIC_SCHEDULE_TYPE is
  PERIOD : duration;
  altimeter_START_TIME1 : duration;
  altimeter_STOP_TIME1 : duration;
  correct_altitude_START_TIME3 : duration;
  correct_altitude_STOP_TIME3 : duration;
  control_elevator_START_TIME5 : duration;
  control_elevator_STOP_TIME5 : duration;
  display_1_START_TIME6 : duration;
  display_1_STOP_TIME6 : duration;
  schedule_timer : TIMER := NEW_TIMER;
  begin
   accept START;
   PERIOD := TARGET_TO_HOST(duration( 5.00000000000000E-01));
   altimeter_START_TIME1 := TARGET_TO_HOST(duration( 0.00000000000000E+00));
   altimeter_STOP_TIME1 := TARGET_TO_HOST(duration( 5.00000000000000E-02));
```

```
correct_altitude_START_TIME3 := TARGET_TO_HOST(duration( 5.00000000000000E-02));
correct_altitude_STOP_TIME3 := TARGET_TO_HOST(duration( 1.25000000000000E-01));
control_elevator_START_TIME5 := TARGET_TO_HOST(duration( 1.250000000000000E-01));
control_elevator_STOP_TIME5 := TARGET_TO_HOST(duration( 2.00000000000000E-01));
display_1_START_TIME6 := TARGET_TO_HOST(duration( 2.00000000000000E-01));
display_1_STOP_TIME6 := TARGET_TO_HOST(duration( 3.00000000000000E-01));
START(schedule_timer);
loop
 delay(altimeter_START_TIME1 - HOST_DURATION(schedule_timer));

 altimeter_DRIVER;

 if HOST_DURATION(schedule_timer) > altimeter_STOP_TIME1 then
   PUT_LINE("timing error from operator altimeter");
   SUBTRACT_HOST_TIME_FROM_ALL_TIMERS
                              (HOST_DURATION(schedule_timer) - altimeter_STOP_TIME1);
 end if;

 delay(correct_altitude_START_TIME3 - HOST_DURATION(schedule_timer));

 correct_altitude_DRIVER;

 if HOST_DURATION(schedule_timer) > correct_altitude_STOP_TIME3 then
   PUT_LINE("timing error from operator correct_altitude");
   SUBTRACT_HOST_TIME_FROM_ALL_TIMERS
                              (HOST_DURATION(schedule_timer) - correct_altitude_STOP_TIME3);
 end if;

 delay(control_elevator_START_TIME5 - HOST_DURATION(schedule_timer));

 control_elevator_DRIVER;

 if HOST_DURATION(schedule_timer) > control_elevator_STOP_TIME5 then
   PUT_LINE("timing error from operator control_elevator");
   SUBTRACT_HOST_TIME_FROM_ALL_TIMERS
                              (HOST_DURATION(schedule_timer) - control_elevator_STOP_TIME5);
 end if;

 delay(display_1_START_TIME6 - HOST_DURATION(schedule_timer));

 display_1_DRIVER;

 if HOST_DURATION(schedule_timer) > display_1_STOP_TIME6 then
   PUT_LINE("timing error from operator display_1");
   SUBTRACT_HOST_TIME_FROM_ALL_TIMERS
                              (HOST_DURATION(schedule_timer) - display_1_STOP_TIME6);
 end if;

 delay(PERIOD - HOST_DURATION(schedule_timer));
 RESET(schedule_timer);
 end loop;
end STATIC_SCHEDULE_TYPE;


procedure START_STATIC_SCHEDULE is
 begin
```

```
    STATIC_SCHEDULE.START;
  end START_STATIC_SCHEDULE;


end autopilot_1_STATIC_SCHEDULERS;


-------------------------------------------------------------------------------------------------
-- Unit          : autopilot_2_static_schedulers
-- Prototype     : CAPS autopilot
-- Date          : June 94
-- Author        : Jim Brockett
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : This package invokes the drivers of the time critical operators in partition 2
-- Notes         : Adapted to the distributed implementation by Jose Carlos Almeida September 1998
-------------------------------------------------------------------------------------------------


package autopilot_2_STATIC_SCHEDULERS is
  procedure START_STATIC_SCHEDULE;
  procedure STOP_STATIC_SCHEDULE;
end autopilot_2_STATIC_SCHEDULERS;


-------------------------------------------------------------------------------------------------
-- Unit          : autopilot_2_static_schedulers
-- Prototype     : CAPS autopilot
-- Date          : June 94
-- Author        : Jim Brockett
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : This package invokes the drivers of the time critical operators in partition 2
-- Notes         : Adapted to the distributed implementation by Jose Carlos Almeida September 1998
-------------------------------------------------------------------------------------------------

with autopilot_2_DRIVERS;  use autopilot_2_DRIVERS;
with PRIORITY_DEFINITIONS;  use PRIORITY_DEFINITIONS;
with PSDL_TIMERS;  use PSDL_TIMERS;
with TEXT_IO;  use TEXT_IO;

package body autopilot_2_STATIC_SCHEDULERS is

  task type STATIC_SCHEDULE_TYPE is
    pragma priority (STATIC_SCHEDULE_PRIORITY);
    entry START;
  end STATIC_SCHEDULE_TYPE;
  for STATIC_SCHEDULE_TYPE'STORAGE_SIZE use 200_000;
  STATIC_SCHEDULE : STATIC_SCHEDULE_TYPE;

  done : boolean := false;
  procedure STOP_STATIC_SCHEDULE is
    begin
      done := true;
    end STOP_STATIC_SCHEDULE;

  task body STATIC_SCHEDULE_TYPE is
    PERIOD : duration;
    compass_START_TIME1 : duration;
    compass_STOP_TIME1 : duration;
    correct_course_START_TIME3 : duration;
    correct_course_STOP_TIME3 : duration;
    control_rudder_START_TIME5 : duration;
```

147

```
   control_rudder_STOP_TIME5 : duration;
   display_2_START_TIME6 : duration;
   display_2_STOP_TIME6 : duration;
   schedule_timer : TIMER := NEW_TIMER;
begin
 accept START;
 PERIOD := TARGET_TO_HOST(duration( 5.00000000000000E-01));
 compass_START_TIME1 := TARGET_TO_HOST(duration( 0.00000000000000E+00));
 compass_STOP_TIME1 := TARGET_TO_HOST(duration( 5.00000000000000E-02));
 correct_course_START_TIME3 := TARGET_TO_HOST(duration( 5.00000000000000E-02));
 correct_course_STOP_TIME3 := TARGET_TO_HOST(duration( 1.25000000000000E-01));
 display_2_START_TIME6 := TARGET_TO_HOST(duration( 1.25000000000000E-01));
 display_2_STOP_TIME6 := TARGET_TO_HOST(duration( 1.95000000000000E-01));
 control_rudder_START_TIME5 := TARGET_TO_HOST(duration( 1.95000000000000E-01));
 control_rudder_STOP_TIME5 := TARGET_TO_HOST(duration( 2.70000000000000E-01));
 START(schedule_timer);
 loop
  delay(compass_START_TIME1 - HOST_DURATION(schedule_timer));

  compass_DRIVER;

  if HOST_DURATION(schedule_timer) > compass_STOP_TIME1 then
   PUT_LINE("timing error from operator compass");
   SUBTRACT_HOST_TIME_FROM_ALL_TIMERS(HOST_DURATION
                                       (schedule_timer) - compass_28_34_STOP_TIME1);
   end if;
   exit when done;

  delay(correct_course_START_TIME3 - HOST_DURATION(schedule_timer));

  correct_course_DRIVER;

  if HOST_DURATION(schedule_timer) > correct_course_STOP_TIME3 then
   PUT_LINE("timing error from operator correct_course");
   SUBTRACT_HOST_TIME_FROM_ALL_TIMERS
                    (HOST_DURATION(schedule_timer) - correct_course_31_37_STOP_TIME3);
   end if;
   exit when done;

  delay(display_2_START_TIME6 - HOST_DURATION(schedule_timer));

  display_2_DRIVER;

  if HOST_DURATION(schedule_timer) > display_2_STOP_TIME6 then
   PUT_LINE("timing error from operator display_2");
   SUBTRACT_HOST_TIME_FROM_ALL_TIMERS
                                   (HOST_DURATION(schedule_timer) - display_2_STOP_TIME6);
   end if;
   exit when done;

  delay(control_rudder_START_TIME5 - HOST_DURATION(schedule_timer));

  control_rudder_DRIVER;

  if HOST_DURATION(schedule_timer) > control_rudder_STOP_TIME5 then
   PUT_LINE("timing error from operator control_rudder");
```

148

```
       SUBTRACT_HOST_TIME_FROM_ALL_TIMERS
                      (HOST_DURATION(schedule_timer) - control_rudder_29_35_STOP_TIME5);
     end if;
     exit when done;

     delay(PERIOD - HOST_DURATION(schedule_timer));
     RESET(schedule_timer);
   end loop;
 end STATIC_SCHEDULE_TYPE;

 procedure START_STATIC_SCHEDULE is
 begin
  STATIC_SCHEDULE.START;
 end START_STATIC_SCHEDULE;

end autopilot_2_STATIC_SCHEDULERS;
```

APPENDIX F. CONFIGURATION FILE

```
-- Unit          : autopilot.cfg
-- Prototype     : CAPS autopilot
-- Date          : September 1998
-- Author        : Jose Carlos Almeida
-- Compiler      : GLADE 1.03p/Gnat 3.10p
-- Description    : This file configures the partitions in the distributed autopilot prototype
-- Notes         : The partitions will be launched manually by the user
```

```
configuration Autopilot is

pragma Version (False);
-- Comes from a GNAT compiler limitation.

pragma Starter (None);
-- The partitions will be launched manually

pragma Boot_Server ("tcp", "sun53:3333");
-- Specify the use of a particular boot_server


Partition1 : Partition := (psdl_streams, altimeter_PKG,
                correct_altitude_PKG, display_1_PKG,
                altitude_command_type_PKG,
                control_elevator_PKG,
                rudder_status_type_PKG,
                elevator_status_type_PKG,
                course_command_type_PKG,
                autopilot_1_drivers,
                autopilot_1_dynamic_schedulers,
                autopilot_1_remote_streams,
                autopilot_1_start_drivers,
                autopilot_1_static_schedulers,
                autopilot_1_streams, gui_pkg);


Partition2 : Partition := (psdl_streams, compass_PKG,
                correct_course_PKG, control_rudder_PKG,
                altitude_command_type_PKG,
                display_2_PKG,
                rudder_status_type_PKG,
                elevator_status_type_PKG,
                course_command_type_PKG,
                autopilot_2_drivers,
                autopilot_2_dynamic_schedulers,
                autopilot_2_start_drivers,
                autopilot_2_static_schedulers,
                autopilot_2_streams, gui_pkg);
```

```
procedure Start_Autopilot is in Partition1;
-- procedure Start_Autopilot will start the Autopilot execution


procedure Start_Partition2;
for partition2'Main use Start_Partition2;
-- procedure Start_Partition2 is the driver procedure for partiton2

Channel_1 : Channel := (Partition1, Partition2);

end Autopilot;
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center...2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library...2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5000

3. Chaiman, Department of Computer Science...1
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5000

4. Professor Man-Tak Shing,..3
   Code CS/Sh
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. Commander M.J. Holden, USN,..1
   Code CS/Hm
   Naval Postgraduate School
   Monterey, CA 93943-5000

6. Diretoria de Ensino da Marinha..1
   A/C Brazilian Naval Commission
   5130 MacArthur Blvd., N. W.
   Washington, DC 20016-3344

7. Centro de Análises de Sistemas Navais...1
   A/C Brazilian Naval Commission
   5130 MacArthur Blvd., N. W.
   Washington, DC 20016-3344

8. Commander Mauricio de Menezes Cordeiro..1
   Instituto de Pesquisas da Marinha
   A/C Brazilian Naval Commission
   5130 MacArthur Blvd., N. W.
   Washington, DC 20016-3344